ACCELERATING MICROARCHITECTURAL SIMULATION VIA STATISTICAL SAMPLING PRINCIPLES

A Dissertation
Presented to
The Academic Faculty

by

Paul David Bryan

In Partial Fulfillment
Of the Requirements for the Degree
Doctor of Philosophy of Computer Science in the
School of Computer Science, College of Computing

Georgia Institute of Technology

May 2013

ACCELERATING MICROARCHITECTURAL SIMULATION

VIA STATISTICAL SAMPLING PRINCIPLES

Approved by:

Dr. Thomas M. Conte, Advisor
College of Computing
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
College of Computing
Georgia Institute of Technology

Dr. Milos Prvulovic
College of Computing
Georgia Institute of Technology

Dr. Gabriel Loh
AMD Research
Advanced Micro Devices

Dr. George Riley
College of Computing
Georgia Institute of Technology

Date Approved: October 12, 2012

To my wife Kelly.

ACKNOWLEDGEMENTS

I first wish to thank my advisor Prof. Tom Conte. His guidance, support, and advice have been invaluable in navigating the Ph.D. waters. Also, his outstanding teaching first gravitated me towards the microarchitecture field, and for that, I am grateful. Thanks also to the entire (tinker) research team during my tenure: Jesse Beu, Rishiraj Bheda, Balaji Iyer, Jason Poovey, Brian Railing, Chad Rosier, and Saraubh Sharma. Over the years, we have had many interesting and lively discussions. We have learned immensely from each other, and I am much better for having the opportunity to work with you all. Go Tinker!

I would also like to thank my wonderful support system: family and friends. To my parents, thanks for your love and support. To my friends, thank you for the same. You have been there, when, at times, the process was almost too much to handle. Without you, this wouldn't have been possible. Thank you all for putting up with me during this time.

I am especially lucky to have a wonderful wife, Kelly. Thank you for moving with me when my research group was relocated to Georgia Tech from NC State. I know the move was especially hard on you, leaving your longtime friends behind. For your sacrifice, I will forever be in your debt.

Thanks to my father, Bill, a bit of a statistics guru, for coaching me early in the process. We had many wonderful debates regarding many of the fundamental statistic concepts covered in this thesis. Thanks to my mother, Carol, for offering encouraging

support. Thanks to my sister, Stephanie, for her confidence and for helping to lighten the mood by poking fun at our parents.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iv

LIST OF TABLES ix

LIST OF FIGURES x

SUMMARY xii

CHAPTER 1: INTRODUCTION 1
1.1 Microarchitectural Design 2
1.2 Simulation Taxonomy 6
1.3 Accelerating Microarchitectural Simulation 10
1.3.1 Workload Reduction 10
1.3.1.1 Reduced Input Sets 11
1.3.1.2 Statistical Simulation 13
1.3.1.3 Program Repetition 16
1.3.1.3.1 SimPoint 16
1.3.1.3.2 EXPERT 29
1.3.1.4 Sampled Simulation 22
1.3.1.5 Summary 24
1.3.2 Simulator Complexity Reduction 25
1.3.2.1 Analytical Models 25
1.3.2.2 Hardware Assistance 28
1.4 Conclusions 32
1.5 Organization of the Document 33

CHAPTER 2: STATISTICALLY SAMPLED SIMULATION 35
2.1 Sampling Basics 35
2.2 Sampling Techniques for Microarchitecture Simulation 41
2.3 Sampling Error 46
2.3.1 Sampling Bias 47
2.3.2 Non-sampling Bias 48
2.4 Statistically Sampled Uniprocessor Simulation 49
2.4.1 Trace-driven vs. Execution-driven Sampling 52
2.4.2 Checkpoints vs. Functional Fast-Forwarding 53
2.5 Warm-up Methods 55
2.5.1 Cache Warm-up Methods 56
2.5.1.1 Excluding Unknown References 56
2.5.1.2 Estimation of Unknown Fill References 57
2.5.1.3 Set Sampling 62
2.5.2 Processor Warm-up Methods 67
2.5.2.1 Sub-component Simulation 67

2.5.2.2 State-reduction Method....68
2.5.2.3 SMARTS....69
2.5.2.4 Minimal Subset Evaluation (MSE)....72
2.5.2.5 Memory Reference Reuse Latency (MRRL)....74
2.5.2.6 Boundary Line Reuse Latency (BLRL)....76
2.5.3 Multi-processor Warm-up....78
2.5.3.1 Memory Timestamp Record (MTR)....78
2.6 Thesis Contributions....82
2.6.1 Reverse State Reconstruction....82
2.6.2 Single-Pass Sampling Regimen Construction....82
2.6.3 Enumerating Challenges that Currently Prevent Multi-threaded Sampling....83
2.6.4 Barrier-Interval Time-Parallel Simulation....84

CHAPTER 3: REVERSE STATE RECONSTRUCTION FOR SAMPLED MICROARCHITECTURAL SIMULATION....85
3.1 Reverse State Reconstruction....85
3.2 Reverse Cache Reconstruction....86
3.3 Reverse Branch Predictor Reconstruction....89
3.4 Experimental Framework....93
3.5 Experimental Results....95
3.6 Related Work....109
3.7 Conclusion....111
3.8 Appendix....112

CHAPTER 4: COMBINING CLUSTER SAMPLING WITH SINGLE PASS METHODS FOR EFFICIENT SAMPLING REGIMEN DESIGN....114
4.1 Statistical Sampling Assumptions....114
4.2 Sampling Regimen Construction....115
4.3 Single-Pass Regimen Design....117
4.4 Simulator Modifications....118
4.5 Profiling Sample....120
4.6 Profiling Analysis....123
4.7 Candidate Selection....125
4.8 Methodology....126
4.9 Results....127
4.10 Related Work....129
4.11 Conclusion....131

CHAPTER 5: OBSTACLES THAT PREVENT ACCURATE AND RELIABLE MULTI-THREADED SAMPLING....132
5.1 Multi-threaded Sampling Obstacles....134
5.1.1 Theoretical Sampling Extensions to Multi-threaded Applications....146
5.1.2 The Lack of Stable Performance Metrics....139

5.1.3 The Circular Dependence Dilemma of Parallel Workload Simulation 141
5.1.4 Relative Thread Progression and Thread Skew 143
5.1.5 Thread Skew Reduction 151
5.2 Conclusion 154

CHAPTER 6: ACCELERATING MULTI-THREADED APPLICATION SIMULATION THROUGH BARRIER-INTERVAL TIME-PARALLELISM 156
6.1 Time-Parallel Simulation 158
6.2 Barrier-Interval Simulation 159
6.3 Experimental Methodology 163
6.4 Results 165
6.4.1 Parallel Simulation Accuracy 165
6.4.2 Error Rates vs. Interval Size 169
6.4.3 Parallel Simulation Speedup 174
6.5 Speedup with Limited Contexts 179
6.6 Related Work 182
6.7 Conclusion 183

CHAPTER 7: CONCLUSIONS AND FUTURE WORK 184
7.1 Contributions 184
7.2 Future Work 186
7.3 Conclusions 189

REFERENCES 190

LIST OF TABLES

Table 1 Transistor Counts of Intel Processors from 1971 to Present 2

Table 2 Descriptive Statistics for Parent and Sampled Distributions 37

Table 3 Confidence Interval two-tail Z-scores with Infinite Degrees of Freedom 40

Table 4 True IPC and Sampling Regimen Data for each Workload 95

Table 5 Warm-up Method Experiments 97

Table 6 Commonly-used Benchmark Suites that contain Barriers 161

Table 7 Architecture Parameters of the Simulated System 165

Table 8 Relative Speedup Efficiency vs. Coefficient of Variation 175

LIST OF FIGURES

Figure 1 VLSI Design Flow 4

Figure 2 Trace-Driven vs. Execution-Driven Simulation 7

Figure 3 Student-t and the Standard Normal Distribution. 38

Figure 4 Cluster Sampling Methodology for Sampled Simulation 50

Figure 5 MSE Cache Specific Warm-up Calculations 73

Figure 6 MTR Updates Caused by Processor Memory Requests 79

Figure 7 Reverse Cache Reconstruction of an Individual Cache Line 88

Figure 8 Reverse State Reconstruction of Branch Table Entries 90

Figure 9 Reverse State Reconstruction for the Return Address Stack 93

Figure 10 RSR Cache Warm-up Results 101

Figure 11 RSR Branch Predictor Warm-up Results 102

Figure 12 RSR Cache and Branch Prediction Warm-up Results 103

Figure 13 RSR vs. SMARTS 106

Figure 14 RSR vs. SimPoint 108

Figure 15 Single-Pass Sampling Regimen Design Flowchart 118

Figure 16 Cluster Sampling Modifications to Enable Derived Subsamples 120

Figure 17 Single Pass Error vs. Sample Size 121

Figure 18 Single Pass Error Rates at High Sampling Rates 122

Figure 19 Single Pass Sample Size vs. Execution Time 123

Figure 20 Single-Pass Sampling Regimen Simulation Speedup 128

Figure 21 Theoretical Extension of Single-Threaded Sampling to Multi-threaded Workloads 138

Figure 22 Formal Definition of Thread Skew 144

Figure 23 Thread Skew Values for *FFT* Benchmark Executions 146
Figure 24 Thread Skew Values for *LU Contiguous* Benchmark Executions..... 147
Figure 25 Thread Skew Values for *OCEAN Contiguous* Benchmark Executions.. 148
Figure 26 Thread Skew Values for *RADIX* Benchmark Executions..... 149
Figure 27 Thread Skew Values for *WATER SPATIAL* Benchmark Executions 150
Figure 28 Barrier Interval Simulation (BIS) 163
Figure 29 BIS Accuracy Measurements (per-workload and average behaviors).... 168
Figure 30 Average Normalized Barrier Interval Sizes..... 171
Figure 31 Average Interval Error vs. Core Counts 172
Figure 32 Interval Errors for *Ocean Contiguous* vs. Core Count 173
Figure 33 BIS Wall-clock Simulation Speedup Measurements..... 176
Figure 34 Accuracy and Speedup Losses vs. Warm-up (1 to 16 cores)..... 178
Figure 35 Accuracy and Speedup Losses vs. Warm-up (32 to 512 cores)..... 178
Figure 36 Limited Context Environment for each Workload 181
Figure 37 Limited Context Environment for the entire Suite 181

SUMMARY

The design and evaluation of computer systems rely heavily upon simulation. Simulation is also a major bottleneck in the iterative design process. Applications that may be executed natively on physical systems in a matter of minutes may take weeks or months to simulate. As designs incorporate increasingly higher numbers of processor cores, it is expected the times required to simulate future systems will become an even greater issue. Simulation exhibits a tradeoff between speed and accuracy. By basing experimental procedures upon known statistical methods, the simulation of systems may be dramatically accelerated while retaining reliable methods to estimate error.

This thesis focuses on the acceleration of simulation through statistical processes. The first two techniques discussed in this thesis focus on accelerating single-threaded simulation via cluster sampling. Cluster sampling extracts multiple groups of contiguous population elements to form a sample. This thesis introduces techniques to reduce sampling and non-sampling bias components, which must be reduced for sample measurements to be reliable. Non-sampling bias is reduced through the Reverse State Reconstruction algorithm, which removes ineffectual instructions from the skipped instruction stream between simulated clusters. Sampling bias is reduced via the Single Pass Sampling Regimen Design Process, which guides the user towards selected representative sampling regimens. Unfortunately, the extension of cluster sampling to include multi-threaded architectures is non-trivial and raises many interesting challenges. Overcoming these challenges will be discussed. This thesis also introduces thread skew, a useful metric that quantitatively measures the non-sampling bias associated with

divergent thread progressions at the beginning of a sampling unit. Finally, the Barrier Interval Simulation method is discussed as a technique to dramatically decrease the simulation times of certain classes of multi-threaded programs. This method segments a program into discrete intervals, separated by barriers, which are leveraged to avoid many of the challenges that prevent multi-threaded sampling.

CHAPTER 1

INTRODUCTION

In 1965, Gordon E. Moore famously predicted the exponential growth rate of integrated circuits in terms of transistor counts. Since that time, microelectronic devices have advanced remarkably from designs containing thousands of transistors to contemporary designs containing billions (see Table 1). Enabled by very-large-scale integration (VLSI) processes, the impact of the semiconductor industry has had far reaching ramifications for many consumer goods ranging from personal music players to cellular telephones, tablets, televisions, laptops, PCs, etc. Computing has become so pervasive that some have argued we have entered the era of ubiquitous computing (ubicomp). The availability of increasingly powerful devices has led to rapid advancements in a wide range of disciplines, including: chemistry, physics, biology, engineering, economics, etc. Over the many generations of different systems and users, the computational ecosystem has evolved towards what may be described as a cat-and-mouse-like situation. Architects create more powerful systems in response to increased demands and, in response to a greater availability of system performance, developers create more resource-intensive applications.

Table 1: Transistor Counts of Intel Processors from 1971 to Present

Processor	Transistor Count	Year
Intel 4004	2,300	1971
Intel 8085	6,500	1976
Intel 8086	29,000	1978
Intel 80286	134,000	1982
Intel 80386	275,000	1985
Intel 80486	1,180,000	1989
Pentium	3,100,000	1993
Pentium II	7,500,000	1997
Pentium III	9,500,000	1999
Pentium 4	42,000,000	2000
Itanium 2	220,000,000	2003
Core 2 Duo	291,000,000	2006
Core i7 (Quad)	731,000,000	2008
Six-Core Core i7	1,170,000,000	2010

1.1 Microarchitectural Design

The design and implementation of a new system[1] typically follows the VLSI design flow, shown in Figure 1. The VLSI design flow describes the various layers of abstraction necessary to create a new physical device. The design flow begins by creating a microarchitectural specification. The specification is created through the satisfaction of product requirements, such as processor features, performance, power budget, transistor budget, and reliability. The behavioral description is then created as a high-level model to represent basic functionality. The high-level model is usually implemented in a high-level language, such as C, C++, or SystemC. During this step, the high-level model is *simulated* with various inputs to determine if the design meets product requirements. If not, then aspects of the microarchitectural specification are

[1] The term system is a high-level abstraction, which encompasses the full computational spectrum from general-purpose processors to embedded systems, FPGAs, ASICs, and custom logic. In this thesis, systems generally refer to processor designs.

revised and reevaluated until the behavioral description is acceptable. Design then proceeds to the register-transfer level (RTL) description, which is a level of abstraction that defines the circuit's behavior as a flow of data between hardware registers, and the logical operations that are performed between them. This step involves the implementation of the design in a hardware description language (HDL), such as Verilog or VHDL. After the HDL implementation has been verified, the RTL model is synthesized (usually with automated CAD tools) to an implementation consisting of logic gates. Logic synthesis generates a netlist containing a full description of all electrical components in the design and their connectivity. The design is converted to a physical layout, representing the circuit as a number of geometric layers of metal, oxide, or semiconductor materials. Finally, the physical layout is sent to a fab (semiconductor fabrication plant) during tape-out.

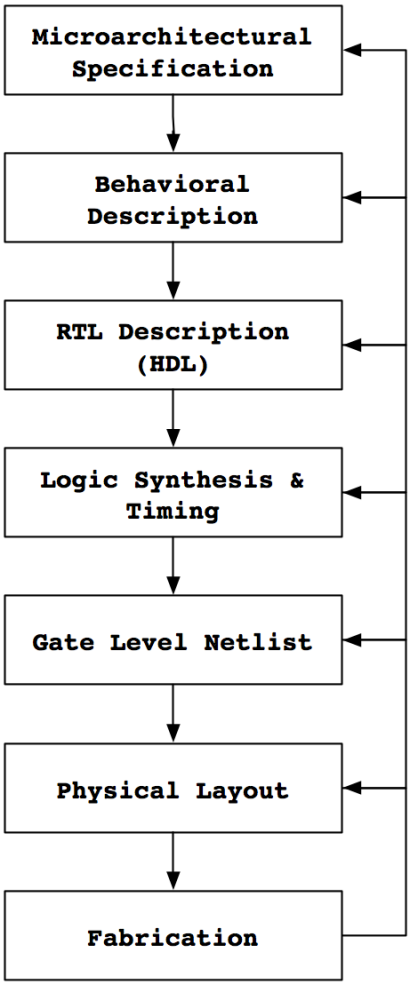


Figure 1: VLSI Design Flow

The VLSI design flow is an example of the iterative design cycle where advancement to subsequent design steps is an exercise in progressive refinement. At many of these steps, simulation is integral towards evaluation. The behavioral implementation is simulated to ensure high-level functionality of available features and to estimate system performance. The HDL implementation is simulated to ensure functionality of module abstractions, and their interactions with one another. The synthesized HDL is simulated until timing closure has been reached. The netlist and

post-layout simulations are performed to collect low-level metrics such as power consumption, parasitic capacitance, etc. Design flaws discovered in a design step cause feedback loops to one or more previous design steps, where the discovery of a flaw in later design steps is more costly. Since device fabrication is very costly and time-consuming (generally requiring six to eight weeks), it is highly desirable for products returning from the fab to be error free. Rigorous simulation and evaluation at all design steps is highly important in attaining a functional and bug-free chip that meets the specified design requirements.

This thesis deals with the front-end of the VLSI design flow between the microarchitectural specification and behavioral description. Between these two steps, architects propose new design features and then use the results from simulation to evaluate them. Even if the new design features have been fully established, simulation is an invaluable resource in the optimization of existing designs. Modern processors are highly complex entities containing abundant design parameters (e.g. pipeline width, memory bandwidth, number of MSHRs, number of cache levels and their organization, cache sizes, NOC parameters, number of processors, coherence protocol selection, etc.). The effects of design parameters may impact system performance, among other things, in non-trivial ways involving interactions that may not be easily or accurately predicted. Given the complex interactions between design parameters, simulations are often required to determine overall system characteristics. Clearly, system design is full of tradeoffs between speed, chip area, power consumption, verification effort, and price. Unfortunately, the abundance of design parameters precludes the search of the entire design space for a globally optimized design. This intractability is made even worse

when considering an optimal design is dependent not only upon the specified microarchitecture, but also upon the workload (and workload inputs).

In any experimental simulation infrastructure, two important properties must be considered: accuracy and stability [26]. Simulation accuracy refers to the fidelity of the simulator: how faithfully does the simulator model the target architecture? If the simulated microarchitecture matches an existing microarchitecture, comparing performance measurements between the simulator and a physical machine may assess accuracy. Direct performance comparisons, such as these, require large programmer effort to validate the simulator. When investigating a microarchitectural feature, that feature is said to be stable if it is shown to demonstrate similar types of performance improvements (or degradation) across a number of different simulators and system configurations.

1.2 Simulation Taxonomy

Simulation broadly defines any method in which system behavior is modeled. For high-level microarchitectural evaluation, various classifications are used to describe the ways in which system modeling is achieved. Levels of simulation detail range from functional to cycle-accurate modeling. Functional simulators (sometimes called emulators) model system behavior at a functional level; they model what is being performed, not how it is being performed. Cycle-accurate simulators model system behavior at a much finer level of detail; they model what is being performed, as well as how it is being performed. These simulators attempt to reproduce the steps taken by a real machine at the cycle level. Simulators may also differ in the extent of the system being modeled, or the simulation scope. Microarchitecture simulators may only consider

the execution of single user-level programs, or subsets of programs for multiprogrammed simulation. In contrast, full-system simulators model the execution of programs running in the presence of an operating system, including peripheral devices.

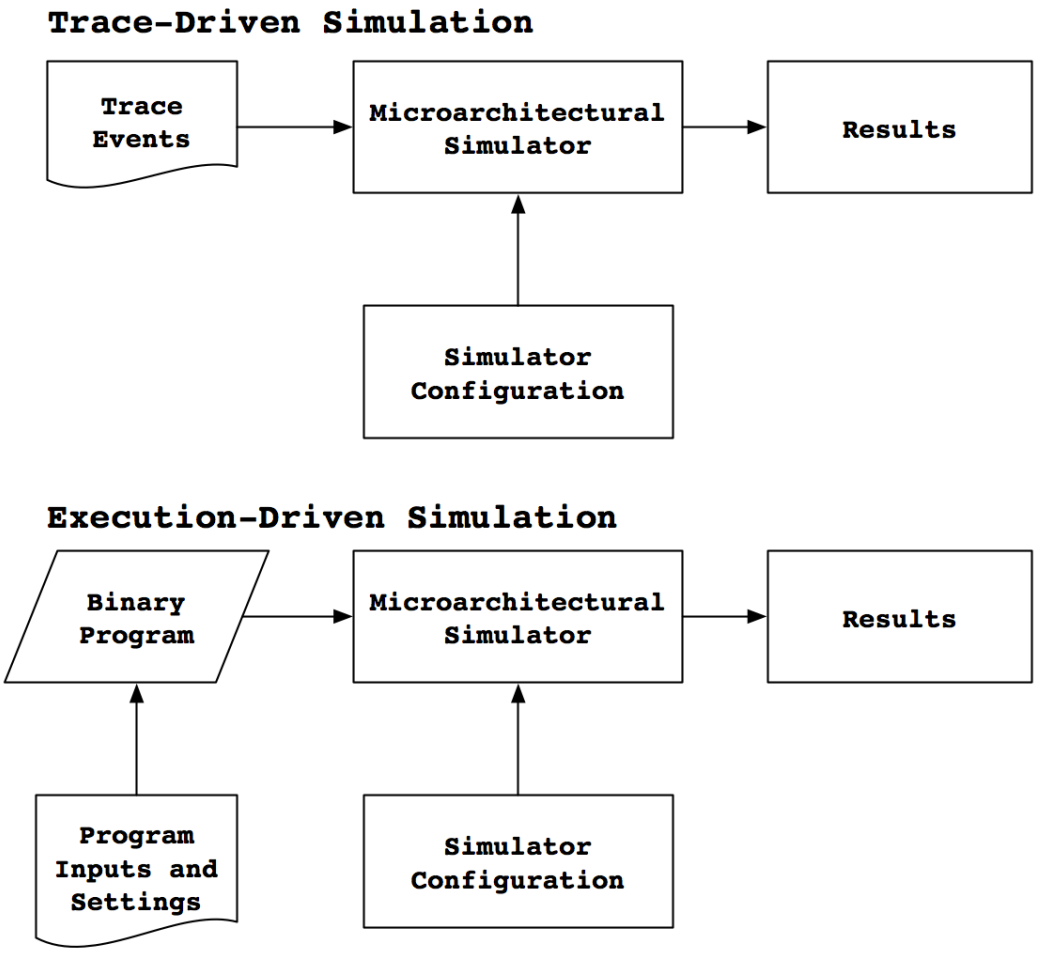


Figure 2: Trace-Driven vs. Execution-Driven Simulation

Simulators are also classified by their input types, which instruct the simulator of desired runtime behavior. Two types of simulator input schemes are trace-driven and execution-driven, and their differences are shown in Figure 2. In trace-driven simulation,

simulator inputs consist of a number of prerecorded trace events. Traces may be collected through functional simulation or by binary or source instrumentation. For cache simulation, a trace would consist of memory references, while a trace of branch outcomes would be used for branch predictor simulation. For many microarchitectural simulators, a trace consists of instructions. Trace-driven simulation incurs a high storage cost due to the disk space required to contain the traces (billions of trace events may correspond to Gigabytes of disk space). Thus, researchers have devised numerous trace-sampling techniques to reduce the storage cost [22], [42], [83]. When leveraging trace-driven simulation, designers must ensure collected traces are appropriate and representative for the evaluation of the target microarchitecture. For example, the use of trace-driven simulation is generally not considered appropriate for the evaluation of multiprocessor systems [48]. In these types of systems, evaluation is highly dependent upon the precise interleaving of memory requests from the individual processors. The observed interleaving is dependent upon many microarchitectural features. If the trace was generated from a host with a different microarchitecture than the one being simulated, then interleaving of trace elements may diverge. Furthermore, traces of memory accesses may include sources of nondeterminism that could artificially bias the evaluation of the target system.

In execution-driven simulation, a binary program is decoded along with program inputs, and the ISA (instruction set architecture) is functionally emulated on demand. Execution-driven environments rely upon a binary program, and so its use necessitates a functional compiler for the ISA under consideration. Since execution-driven simulators gather simulation inputs from the binary itself, the storage requirements of workloads are

significantly less than trace-driven inputs. Additionally, execution-driven simulators are capable of the detailed modeling of speculative paths. For example, branch predictors (and branch target predictors) are commonly utilized to reduce pipeline stalls due to control flow. During simulation, incorrectly predicted branch outcomes or branch targets cause pipeline fetches to instructions along the wrong-path until the misprediction is discovered and corrected. Complex processor designs with longer pipelines tend to incorporate better branch predictors since their associated misprediction penalties are higher.

Execution-driven simulation environments have become increasingly sophisticated. In addition to modeling out-of-order execution, these simulators commonly incorporate advanced speculation techniques throughout the pipeline. The cumulative effect has resulted in simulation environments orders of magnitude slower than native execution [33], [34]. At the same time, benchmarks have also exploded in dynamic instruction counts in order to stress next-generation systems. For example, the largest SPEC2000 benchmark [74] using the reference input set contains hundreds of billions of instructions. In contrast, the largest SPEC2006 benchmark with reference input set contains trillions of instructions. Execution of a workload that can be performed in minutes natively can take weeks, months, or longer to simulate [33], [34], [56] (even a simple Hello World program written in C contains tens of thousands of instructions). This phenomenon is exacerbated even further when other components are considered, such as full-system simulation (including system calls, process scheduling, semaphore management, etc.), and their extension to multi-/manycore systems. The combination of

higher simulation complexity, as more events are modeled, and larger benchmarks make the simulation of entire workloads intractable.

1.3 Accelerating Microarchitectural Simulation

The speed of simulation is of particular importance to designers and researchers. Faster simulation allows for more design features to be explored, and facilitates searching larger portions of the design space. Since the complete simulation of large workloads is infeasible, it is common for researchers to only simulate a small subset to reduce simulation times. Unfortunately, it is also common for researchers to execute a single instruction window located after initialization code in a benchmark. Although effective in reducing simulation time, the arbitrary selection of instructions is not guaranteed to be representative and can lead to inferences that are misleading or inaccurate [23], [30], [50]. In fact, one study suggests blindly fast-forwarding to arbitrary simulation points may lead to average errors of up to 80% [73]. In order to address this problem, many techniques have been proposed to reduce the simulation times while preserving accuracy. A cross-section of the various methodologies and techniques (related work) is presented, categorized by the manner in which acceleration is achieved.

1.3.1 Workload Reduction

In the evaluation of any new idea or feature, benchmark performance determines the overall effectiveness of a proposed change. Since it is prohibitively time-consuming to simulate complete workloads, one solution is to reduce their sizes. An input program is passed to a reduction function that outputs a much smaller program. If the output program is representative of the input program, then simulation results of both should be similar.

1.3.1.1 Reduced Input Sets

Program binaries often rely upon the use of inputs to dictate parameters or datasets to be used during an execution. Many benchmark suites used for experimental simulation provide various predefined input sets. The choice of input set is typically dependent upon the type of activity being performed by the designer. Small inputs are useful for verification and debugging, and are designed to run in a few minutes. Medium inputs are useful for more detailed simulator testing and the collection of preliminary data, and are designed to run in a few hours. Large inputs are useful for final performance collection, and are intended to provide realistically sized working sets. Simulating with large inputs may require a few days, or longer, to complete. Often, benchmark suites also provide native inputs for execution on real hardware. For example, the parsec benchmark suite provides the input sets: test, simdev, simsmall, simmedium, simlarge. The SPEC benchmark suite provides the input sets: test, train, and reference.

A typical approach to shorten runtimes involves using a smaller predefined input set (when available), or the manual reduction of inputs. Unfortunately, the naive modification of input sets may drastically alter the execution profile of a program [46]. Benchmarks are designed to stress various parts of the microarchitecture (e.g., branch predictor, cache hierarchy, functional units, memory subsystems, etc.). If the dynamic profile is significantly altered, then the benchmark is no longer guaranteed to realistically test the system components for which it was designed. Thus, performance measurements taken from modified input sets may not be representative of the original program.

KleinOwoski, et al. [46] addressed this problem by investigating ways to accurately reduce the runtimes of SPEC2000 workloads (using reference inputs) into three distinct sizes: small (SmRed), medium (MdRed), and large (LgRed). Workload reduction techniques varied on a per-benchmark basis. For some benchmarks, the input sets were reduced. For other benchmarks not utilizing input files, reduction involved modifying the benchmark source code to reduce loop iterations. After the workload reduction, various benchmark characterization metrics (e.g., IPC, instruction mix, and memory behavior) were compared with the unmodified program to assess reduction accuracy. Accuracy was determined by calculating chi-squared, χ^2, statistics for each program, and by comparing function-level profiles. A chi-squared statistical test was used to test goodness-of-fit, by testing the similarity between sample data and the parent population. In this study, a chi-squared confidence interval was calculated using a 90% critical value.

Although reduced workloads generally mimicked the behavior of the original, some benchmark characteristics were not invariant after the transformation. For example, significant discrepancies were observed between the instruction mixture results of the reference input set and LgRed. The observed discrepancies between the original program and the reduced program highlight the challenges associated with workload reduction. Although reduced program inputs can generate programs that are significantly faster to simulate, they cannot guarantee representativeness with respect to the original program. In other words, the only method to ensure the smaller program and the original have similar characteristics is to simulate both.

1.3.1.2 Statistical Simulation

Statistical simulation is a technique used to reduce simulation times by reducing the sizes of simulation inputs (i.e., the "workload" being simulated). Statistical simulation is also sometimes called synthetic trace generation, and is performed via the following three steps. First, simulations are conducted to collect a number of key program characteristics and are used to generate a *statistical profile* of the workload. Second, information gathered in the statistical profile is used to generate a *synthetic workload*. Third, the synthetic workload is simulated, which is much smaller than the original workload.

The goal of statistical simulation is to generate a synthetic trace that captures the behavior of the original program (which is usually less than a few million instructions). Thus, performance characteristics measured by simulating the synthetic trace should closely resemble the original workload. The statistical profile should not be overly complicated (limiting the number of distributions collected), while preserving the accuracy of the performance prediction [32].

When performing statistical simulation, a number of important questions must be answered, such as: What characteristics should be considered during the initial statistical profile of the workload? How can these statistical profiles be synthesized to generate representative and accurate synthetic traces?

The statistical profile contains a number of important program behavior distributions used to generate the synthetic trace, which are gathered through cycle-accurate detailed simulation [65]. Frequently measured characteristics using this technique include: instruction mixtures, inter-instruction dependencies (through registers

and memory locations), statistical control flow graphs (transition probabilities between basic blocks), per-branch misprediction rates, and per- load/store cache miss rates, etc. [27], [32], [65], [66]. Commonly, instruction dependencies are measured using a dependence matrix, which tracks the distance between two dependent instructions as a function of the instruction type. Dependence checking may predict the probability that prior instructions are dependent ("upstream dependencies" [27]) or future instructions are dependent ("downstream dependencies" [65]).

Instructions in the synthetic trace generally contain the following information: the instruction type (opcode) percentages, the number of source operands per instruction type, the inter-instruction dependencies for each source operand, and locality information. The locality information includes instruction cache miss information, data cache and TLB miss information (for loads), and branch misprediction information. For example, upon generation of a load within the synthetic trace, the locality information would describe if that load is an L1 D-cache hit, L2 miss or hit, or if a TLB miss was generated.

While simulating the synthetic trace, simulation inputs (e.g., trace events) are generated randomly from the probability distributions collected during statistical profiling. Additionally, previously recorded locality information is used to probabilistically determine if specific memory instructions are misses or hits within the cache hierarchy, or if branch instructions are correctly predicted. Generally, simulation of the synthetic trace utilizes a simplified version of the simulator (the pipeline is still modeled, but there is no need to compute values, store results, or model the memory hierarchy). Simulation then proceeds until performance metrics converge (e.g., until the

standard deviation is less than 1% [65]). Eeckhout et al. [27], however, argue synthetic traces should be syntactically correct to allow the synthetic traces to be simulated using existing simulators.

Statistical simulation is much faster than a full cycle-accurate simulation of the original program (e.g., speedups of 40x to 70x have been reported [32]), and may be used to rapidly explore the design space in early design stages. Compared with sampled simulation, statistical simulation requires fewer instructions to be simulated [32]. Additional applications of statistical simulation include: processor evaluation, system evaluation, program characterization (the isolation of program characteristics that affect system performance), physical design, and high-level power modeling, etc. [65]. However, statistical simulation is not meant to replace cycle-accurate simulation. Conversely, statistical simulation should be used to coarsely evaluate various design points. Cycle-accurate simulation may then be used to fully investigate design points of interest.

Although effective in reducing simulation times for many workloads, statistical simulation has certain limitations. The collection of program characteristics relies upon detailed cycle-accurate simulation, which requires a full simulation of the workload under test for the microarchitecture of interest. Since locality probabilities are required as an input to this technique, locality information must be recomputed if the memory hierarchy or branch predictor is changed, thereby requiring another full detailed simulation of the original workload. Additionally, certain program behaviors may not be easily reproduced through Monte Carlo methods (e.g., errors in excess of 20% have been observed [65]). Although iterative refinement of conditional probability distributions may decrease

observed error rates, their original derivation and subsequent use are largely empirical. Others have proposed methods for estimate trace representativeness, such as Iyengar et al. [37], however, the proposed R-metric makes unrealistic assumptions (e.g., infinite caches).

1.3.1.3 Program Repetition

Workload reduction has also been proposed by exploiting program repetition. For example, although SPEC2000 only contains static instruction counts in the hundreds of thousands, their executions contain dynamic instruction counts in the tens or hundreds of billions (implying large-scale repetition [56]). Programs are inherently repetitive; execution similarities are observed across loop iterations or repeated function calls at the microarchitectural level [56]. If regions of program similarity may be identified, then their simulation may be avoided altogether. For example, consider a sinusoid with a given period *T*. Using only the knowledge of a single period, all attributes associated with the wave may be characterized. Large-scale program repetitions, when identified, may be thought of as a form of memoization applied at the hardware level.

1.3.1.3.1 SimPoint

A popular approach to workload reduction utilizes SimPoint, and was proposed by Sherwood, et al. [73]. The objectives of the SimPoint tool are to significantly reduce simulation times by selecting small, representative sections of the program for detailed simulation. Assuming the simulated regions (or points) characterize the behavior of the program, then performance measurements should also be representative.

The SimPoint approach is performed as follows. For each program-input pair, functional simulation is performed to quickly gather BBV (basic block vector) signatures

at 100M instruction intervals over an entire execution of a program. A BBV signature consists of a one-dimensional array that contains an element for every static basic block in the program; elements contains counters indicating how many times a basic block has been entered during an interval. Each element is multiplied by the number of instructions contained in the basic block, and normalized to indicate the proportion of time spent in each basic block during an interval. The dimensionality of each BBV signature is reduced to 15 dimensions using a random linear projection (clustering is exceedingly difficult at higher dimensionality). K-means clustering is a method in which *N* observations are partitioned into *k* distinct clusters, where each observation is assigned to the cluster with the closest mean. The purpose of k-means in this context is to partition observations into *k* distinct program phase classifiers, each containing similar BBV signatures. The k-means clustering algorithm is repeatedly performed using values of k between 1 and 10. Results of each value of *k* are ranked via a Bayesian Information Criterion (BIC) value. BIC is commonly used statistical method for model selection; models having higher BIC values are "better" than those with lower values. The k-means clustering with the lowest value of *k* within 90% of highest obtained BIC value is chosen. The rational for selecting lower values of *k* (when possible), is that it may reduce simulation times since it reduces the number of simulation points.

The effectiveness of SimPoint is demonstrated by comparing IPC measurements, branch misprediction rates, and cache miss information across a number of different workloads. When using multiple simulation points (values of k greater than 1), average IPC error rates were less than 3%. SimPoint is microarchitecturally independent, since it relies upon functional information to perform the BBV analysis and

classification. Thus, SimPoint analysis needs only to be performed once per program-input pair. The reliability of SimPoint across various microarchitectural configurations was shown in [70]. In this study, Perelman, et al. proposed an extension to SimPoint which preferred the selection of simulation points that occurred earlier in the dynamic instruction stream to significantly reduce functional-skipping costs. As the authors note, functional-skipping may require up to several days to reach a simulation point located deep into an execution.

One of the greatest benefits of SimPoint is its ease of use. Researchers and designers may obtain the distributed toolchains for popular simulation environments [14], which perform the BBV signature collection and clustering algorithms. These toolchains provide designers with automatic reporting of simulation windows; they are largely free from the burdensome prospect of independently learning, implementing, and verifying the SimPoint technique for their own simulation infrastructures. While effective, critics of SimPoint note the heuristic by which the regions are selected utilizes systematic sampling. Since the probability of selection is not random, statistical tests such as the confidence interval may not be reliable.

An extension to SimPoint, called Variance SimPoint [70], has been introduced to calculate error bounds for sampled clusters. Such error bounds can be calculated if SimPoint selects clusters of execution at random. SimPoint has also been extended by Van Biesbrouck, et al. [76] to identify of representative co-phases for the simulation of multiprogrammed workloads on SMT microarchitectures.

1.3.1.3.2 EXPERT

Liu and Huang [56] propose EXPERT, a related but novel technique in reducing workload sizes. Rather than partitioning workloads based upon fixed size dynamic instruction windows (as in SimPoint), EXPERT performs the partitioning at the software level. The purpose of partitioning based upon software constructs is to reduce the likelihood of false positives, since periodicity is captured more naturally than the arbitrary selection of fixed window sizes. For example, two unrelated code sections may both have low IPC, whose root causes may be different; it is desirable these two code sections not be classified as similar since their behaviors may not be robust to microarchitectural changes. In other words, changing the microarchitecture may cause two similarly behaving sections to behave differently. The proposed scheme has many similarities with sampled simulation, and may be considered a form of stratified sampling.

EXPERT consists of the following systematic approach to workload reduction. Programs are partitioned into static code sections based upon subroutines and outermost loops. Using profiling results, subroutines whose dynamic instruction counts are less than 0.5% of the total execution are considered moot and are excluded from a sample. Additionally, two thresholds control the selection of remaining sections. Sections less than 50,000 instructions are considered too short for reliable measurements in isolation since they likely contain high non-sampling bias. Small code sections are treated as extensions of the caller's execution. Sections longer than one million instructions are classified as long sections. For long code sections containing loops, sampling is also

applied to loop iterations if the sampling unit sizes are greater than one million instructions.

After partitioning, all important code section instances are characterized by measuring the coefficient of variation (CV) for a range of metrics. This process requires a detailed simulation over the entire workload under test. To ensure characterization is only performed over the code sections of interest, instrumentation is added to track when control flow enters or exits a relevant section. Since the complete simulation of SPEC2000 workloads is intractable with reference input sets, the authors perform characterization using multiple training inputs and microarchitectural configurations. CV values for metrics are used to assess variability within code sections; the authors call this measurement the variation factor. The variation factor controls the sampling rate for future simulations, which is insensitive to changes to the input set or microarchitectural changes. Workload partitioning and characterization may be considered equivalent to stratification in stratified sampling.

Each code section is systematically sampled, based upon program repetitions observed during characterization. Identified areas of measurement are simulated in full detail, and functional skipping is used to transition between measurement sites. SMARTS functional warming [84] is used to warm cache and branch predictor state while skipping (see Chapter 2). Two approaches of sampling are proposed: with preprocessing and without preprocessing. When preprocessing is utilized, the sample size, n, is calculated from the variation factor according to a desired confidence level and margin of error. The preprocessing approach also requires an additional pass of the workload in order to obtain the population size, N, in number of sections. Detailed

measurements occur every *N/n* sections; *a priori* knowledge of the population size also ensures measurements are equally spaced across the entire execution. Without preprocessing, the size of the workload is unknown and measurements obtained from n sections may not be spread throughout the execution. In this case, a base sampling rate is used to obtain measurements throughout the program, and may result in over-sampling. Obtained measurements for each code instance are combined using a weighted mean to obtain program-wide statistics.

The EXPERT methodology significantly reduces the runtimes of SPEC2000 benchmarks from hundreds of hours to less than 10 hours. The average CPI error rate is 0.9% and the average error for a variety of other metrics is 3.8%. With preprocessing, obtained approximate speedups ranged from 10x to 100,000x. Without preprocessing, obtained approximate speedups ranged from 4x to 15x. Although effective, the proposed technique has certain drawbacks. In order to maximize the accuracy of the stratification process, the input sets and microarchitectural configuration used for partitioning and classification should match those of the systematically sampled simulation. Unfortunately, doing so would erode all possibility of accelerating the simulation. Although changing the input sets and microarchitectures used for stratification were an effective approximation for SPEC2000 workloads, the modification of input sets is not guaranteed to be an accurate approximation of other arbitrary target workload-input pairings. Additionally, the application of systematic sampling to approximate random sampling should only be relied upon in the absence of population periodicity (the characteristic which EXPERT exploits for acceleration). Finally, confidence intervals were also not calculated to assess the reliability of sample estimates.

1.3.1.4 Sampled Simulation

One effective and accurate technique in workload reduction involves the application of statistical sampling to microarchitectural simulation. Rather than simulating an entire workload in full detail, sampled simulation strives to identify representative elements of the workload to represent the behavior of the whole. Measurements obtained from individual elements (sampling units) may be combined to form accurate program-wide estimates[2], assuming associated sampling and non-sampling bias have been reduced. A thorough explanation of fundamental statistics concepts, sampled simulation procedures, the reduction of bias, and its context within microarchitectural simulation are discussed in Chapter 2. Sampled simulation is the main subject of this thesis.

Since sampled simulation is based upon statistical theory, this method of workload reduction provides a number of key advantages. Unlike many of the aforementioned techniques, the reliability of an estimate may be based upon sample data alone. In many of the previous techniques, the only method to evaluate estimate accuracy involved their direct comparison with the full execution. This was due to various violations of fundamental assumptions that must be satisfied in order for sampling theory to be correctly applied. Furthermore, the use of statistical sampling also traded accuracy for speed, but did so in a controlled manner. Although error was introduced (the difference between a random variable's expected value and the true population mean), it could still be bounded using statistical constructs such as the confidence interval.

---

[2] Population estimates of sampled data generally encompass metrics that may be mathematically combined by computing a mean. Other resampling techniques, such as the jackknife and bootstrap [68] may be used to estimate rate-based metrics.

With so many benefits, one may wonder why sampling techniques have not dominated simulation methodologies. Indeed, some members of the microarchitectural community have argued that integrating statistical rigor will improve the design evaluation process by increasing the confidence in simulation results [89]. Unfortunately, statistically rigorous simulation methodologies are not commonly used in computer architecture research [89]. One possible reason for this phenomenon is many computer engineering/computer science curricula do not heavily focus upon statistics coursework beyond the introductory level. Furthermore, the use of sampling introduces additional complexities into the simulation framework that are typically not implemented in stock simulators. Simulators that incorporate sampling techniques must implement various levels of simulation fidelity and provide mechanisms to correctly transition between them. Slower cycle-accurate simulation is used to model population elements. Fast functional-warming (FFW) techniques rely upon the instantaneous application of specific emulator data (e.g., branch outcomes and predictor tables, cache tag-store values, etc.) to various simulator components. Optional FFW checkpoint optimizations require the storage and retrieval of architectural checkpoint files. Usage of detailed warming phases may require precise resetting of system statistics to avoid measurement contamination by non-sampling bias. Providing dynamic transitions between these simulation phases may involve non-trivial modifications to the simulator that are required to maintain simulation correctness and stability (e.g., the reclamation of dynamic resources after detailed measurement to avoid memory leaks). The number of non-trivial simulator modifications required by sampled simulation has caused other techniques such as SimPoint to gain community traction more quickly.

1.3.1.5 Summary

A cross-sectional survey of related work on the acceleration of simulation though workload reduction techniques was presented in order to evaluate their strengths and weaknesses. Each technique provided designers with clear benefits by trading simulation accuracy for speed. Furthermore, all workload reduction schemes provided clear empirical evidence demonstrating the accuracy. Only two schemes, however, were currently based upon theoretical foundations: EXPERT and sampled simulation. All workload reduction techniques encompassed a reduction function that reduced the target workload to a smaller, more reasonably sized, substrate for simulation. Without the basis of statistical theory or some other theoretical foundation, the reduction function trades accuracy for speed in an uncontrolled manner. Thus, the only way to ensure representativeness of the reduced workload—and by extension, simulation accuracy—involves the simulation of the original (intractable) workload for comparison. Indeed, many of the proposed techniques occasionally exhibit sporadic outliers with high errors. If design decisions are based on these data points, then incorrect conclusions could be drawn.

EXPERT and sampled simulation are both based upon sampling theory. EXPERT relies upon stratified sampling techniques, and sampled simulation relies upon cluster sampling. Both strategies have been extensively studied, and have been successfully utilized across a variety of disciplines. However, both sampling strategies are based upon fundamental statistical assumptions, which must be strictly followed for population inferences to be valid. The violation of statistical assumptions is just as

dangerous as reliance upon empirical techniques, since in either case wildly inaccurate conclusions may be drawn.

1.3.2 Simulator Complexity Reduction

An orthogonal approach to workload reduction in accelerating simulation involves reducing the complexity of the simulator. Workload reduction curtails the program length to obtain faster measurements; it reduces the input to the simulator. Simulator complexity reduction diminishes the amount of modeled work performed for each input; it may reduce the simulator's effort given a fixed input or remove the need for a simulation entirely.

1.3.2.1 Analytical Models

An analytical model is a set of equations used to describe the performance of a computer system. The developed model may be used as an alternative to detailed simulation and has clear benefits with respect to runtimes. System evaluation using models are much faster than detailed simulation. Detailed simulation is extremely accurate and generates considerable data for analysis, but does not provide designers with insights regarding system behavior. Design parameters often have complex interactions with one another, where an intuitive understanding of their impact upon performance is difficult [39]. Analytical modeling can help to provide designers with these insights. A variety of models exist, including: Markov chains, artificial neural networks (ANNs), linear and multiple regressions, etc. These techniques generally rely upon the mapping of predictor (or explanatory) variables to system response variables. The construction of models typically requires domain-specific knowledge in order to isolate and identify effective predictor variables. Data obtained through detailed simulation are used to

provide training data necessary for model construction. After the collection of training data, model construction and subsequent evaluation are efficient due to highly optimized numerical linear algebraic libraries. Analytical modeling may be considered a form of statistical simulation, applied to the simulator rather than the simulator inputs.

Karkhanis and Smith [41] derived an analytical modeled to predict performance of a superscalar processor. Their model consisted of two components. The first component determined idealized performance that could be obtained in the absence of miss-events. Miss-events were defined as events that inhibited ideal throughput, and consisted of branch mispredictions, instruction cache misses, and long latency data cache misses. The second component determined the performance losses incurred by miss-events with respect to system throughput. Linear models were developed for each individual type of miss-event using data obtained from trace-driven simulations. The model was demonstrated to have an average error of 5.8% and a maximum error of 13% for a number of SPEC2000 workloads. Noonburg and Shen [64] utilized Markov chains to model superscalar performance. Regression models have also been extended to predict the performance of multiprocessors [53].

Lee and Brooks [52] used analytical models to accelerate design space searches. The design space explored in this study consisted of approximately 22 billion configurations-- 1 billion design parameter combinations coupled with 22 benchmarks. Using data obtained from 4,000 simulations randomly selected from the design space, regression models were constructed to predict the performance and power of arbitrary points in the design space. Regression models were constructed for a number of predictor variables, using methods of least-squares fitting. In the event predictor

variables were highly correlated[3], the affected terms were combined to form hybrid predictor variables. Non-linear relationships between response and predictor variables were handled through polynomial transformations. During the fitting of these relationships, piecewise polynomial spline functions were utilized incorporating a variable number of knots for integration. Once the full model was developed (incorporating all predictor terms), the effect of individual predictors was evaluated to obtain an accurate model with the least number of terms. The models with and without a particular predictor were compared using a statistical F-test to identify the best model which fits the data. Median error rates for the developed models were 4.1% for performance predictions, and 4.3% for power predictions. The maximum error outliers ranged between 20% and 33%.

Joseph et al. [39] also used regression modeling to predict processor performance, but removed the domain-specific knowledge necessary for the selection of relevant predictor variables, as was required in [52]. An iterative procedure was proposed that incorporated Akaika's Information Criterion (AIC) for model selection and D-optimal experimental design to guide the selection and incorporation of predictor variables. The goal of this technique was to automate the discovery a model that fits well, utilizing a minimum number of parameters. Restricting the number of parameters in regression models is useful in avoiding model over-fitting. Model over-fitting causes the predictive power of a model to be exaggerated. Although such models may predict

[3] Multicollinearity is a statistical phenomenon in which two or more predictor terms are highly correlated. Although multicollinearity does not hinder the predictive power of the multiple regression models, it may prevent software packages from performing the matrix inversion necessary to compute regression coefficients.

outcomes from the training dataset with impressive accuracy, they do not generalize well when applied to other inputs.

Analytical models are powerful tools in the evaluation of systems; once constructed, they may even be used in lieu of detailed simulation. Unlike workload reduction techniques that serve to reduce simulation times, they cannot reduce the number of simulations necessary to explore the design space. Many have noted analytical modeling should not be considered as a method to avoid simulation, but rather as a complementary technique. Developed models may be used to coarsely characterize regions of the design space, and areas of interest may be investigated more fully through detailed simulation. If the accuracy of the developed models are high, then analytical modeling can dramatically reduce the number of simulations required. Unfortunately, the complexities of many systems have limited modeling accuracy [41]. More importantly, there is currently no technique to evaluate model accuracy without performing the detailed simulation of the original workload.

1.3.2.2 Hardware Assistance

Movement along the computational spectrum involves tradeoffs between speed and generality. Custom logic and application specific integrated circuits (ASICs) perform computation at the fastest speeds, but are the least flexible. General-purpose processors excel in their flexibility, but at a cost of some performance. In this middle of this spectrum exists field programmable gate arrays (FPGAs), which offer excellent flexibility and good performance. One mechanism to alleviate the modeling burden is to off-load computational tasks from the simulation infrastructure to be performed by fast hardware units. If the simulated architecture (the target) is sufficiently similar to the

system executing the simulator (the host), then certain modeling tasks may be performed through direct execution [24]. For example, a target's floating-point multiplication instruction could be executed as a host instruction. The time required to perform this calculation in hardware could then be substituted for the simulated time, and simulation tasks may be reserved for operations that are unavailable from the host. If the host and target share a common ISA, then functional emulation may be performed entirely by the host. Direct execution is orders of magnitude faster than simulation, and has been incorporated into simulators such as the Wisconsin Wind Tunnel II (WWT II) [63] and Graphite [62]. Alternatively, certain modeling tasks may also be performed with co-execution on FPGAs. The increase in computational power and speed of FPGAs has created renewed-interest in FPGA-based solutions [18], and has led to projects such as the RAMP initiative [4].

Chung, et al. [18], [19] proposed the PROTOFLEX framework, a hybrid simulation environment based upon Virtutech Simics [60], leveraging FPGAs to accelerate simulation. The authors focused on performance-dominating operations that contribute to the majority of simulation runtime, and transplanted these operations onto FPGA devices. To simplify the HDL necessary for FPGA execution, micro-transplants were leveraged whereby only subsets of functionality were implemented. If unimplemented instructions or events were encountered during execution (e.g., TLB misses, complex instructions, etc.), execution could be deferred to simulation kernels running on nearby embedded processors. Additionally, the embedded kernels reduced expensive overheads associated with data transfer between the host system and the FPGA device, as well as the latencies incurred waiting for the execution of the slower software

simulator. Linear programming techniques were incorporated to optimize the mapping strategies of each operation to a particular computing fabric (host system, embedded processor, or FPGA).

Chiou, et al. [17] proposed FAST, an alternative FPGA-based simulation methodology. Execution-driven simulations of target microarchitectures generally involve two components: a functional model that emulates the ISA and a timing model that consumes functional data. Since the timing model dominated simulation times (and are inherently more parallel than the functional model), the authors chose to offload the entire timing model for FPGA implementation. The functional model provided correct-path instructions to the timing model for processing. Timing model branch mispredictions resulted in control messages requesting the functional model to provide the necessary wrong-path instructions. Although every branch misprediction resulted in expensive communication between the FPGA and functional models, the authors noted branch prediction were generally very accurate. To ensure memory orderings were consistent between the functional and timing models, each read operation was annotated with true dependency information, and each write operation was annotated with output dependency information. During simulation, inconsistent read/write orderings caused the functional model to rollback and re-execute instructions in precise order. The timing model in this study modeled a uniprocessor superscalar architecture, implemented onto a single FPGA. By offloading the entire timing simulator to an FPGA, the authors achieved high simulation throughput (1.2 MIPS, on average).

Hardware assisted execution allows designers to leverage fast hardware to accelerate subsets of the simulated target operations. If the target and host machines are

sufficiently similar, then direct execution can provide a faster alternative to software-only evaluation. (This requires the designer have access to native hardware similar to the one being designed.) In many contexts, the requirement of similarity is not overly constraining, since many next-generation systems are based upon the current generation. However, if the design is experimental in nature and differs significantly, or if the designed system is different from the host machine (e.g., has a different ISA), then direct execution cannot be utilized.

The implementation of simulated components to FPGA devices has also been used to considerably reduce simulation times. In contrast to the other techniques, FPGA execution does not tradeoff speed for accuracy, but rather speed for development times. FPGA products contain programmable logic blocks connected by configurable interconnection resources. The method of programming FPGAs requires much lower level operational details than does a high-level language implementation, and requires simulation tasks be implemented using synthesizable RTL. The development effort associated with this translation may be very high, depending upon the complexity of the microarchitecture. One of the main benefits of implementing simulation models in high-level programming languages is the relative ease (compared to HDL implementations) in which all aspects of the microarchitecture may be modified. In the early stages of the behavioral specification, many design alternatives are considered, some requiring non-trivial microarchitectural changes. Although RTL descriptions are eventually necessary for fabrication, their construction is more time-consuming than their high-level counterparts. Thus, non-trivial microarchitectural changes require much higher development times than similar changes in a software-based simulator. The premature

translation of the behavioral specification would require RTL for every preliminary design alternative. Ultimately, if it is too difficult to modify a simulator, microarchitects will evaluate fewer alternatives and make poorer decisions [69].

1.4 Conclusions

Researchers and designers rely upon the results of simulation to make design decisions. The reduction of simulation times is useful in reducing the iterative design cycle, allowing more design alternatives to be investigated, permits larger areas of the design space to be explored, and results in higher quality products. Simulation is a major bottleneck in this process. Historically, uniprocessor simulation has become faster simply through the realization of increasingly powerful host systems that leveraged smaller features sizes and the extraction of ever-higher levels of instruction level parallelism (ILP). Unfortunately, the "power wall" has necessitated the abandonment of previous strategies that incorporated increased speculation techniques and deeper pipelines to attain faster clock speeds (faster operational clock speeds create increased heat densities, that must be dissipated). The ushering in of the manycore era has resulted in systems that now target thread level parallelism (TLP) to extract higher system performance. As a result, the performances of individual processors are not expected to continue their rapid pace. In other words, the executions of sequential simulators on next-generation systems are not expected to attain similar speedups that were historically achieved solely through their execution on new systems.

The complexity of new systems is also currently outpacing simulation technologies. When simulating multi-/manycore systems in a sequential fashion, there exists a proportional relationship between the number of cores and simulated slowdowns

[19]. These trends, therefore, suggests accelerative strategies will only become more important in the future if processor scaling continues. Many efforts have been made to parallelize simulation environments through techniques such as parallel discrete-event simulation (PDES). Prior attempts to parallelize software simulators have been met with limited success [19], due to the communication overheads between the highly coupled components. The challenges associated with multi-/manycore systems are discussed in Chapter 5.

1.5 Organization of the Document

The techniques described in this chapter are meant to place the subject of this thesis (sampled simulation) into the proper context within the landscape of simulation technologies. Its purpose is to provide an overview and classification of current practices to accelerate simulation, as well as to describe the advantages and disadvantages of each technique. The structure of the remainder of this thesis is as follows: Chapter 2 provides a detailed description of sampled simulation, its foundation in statistical theory, the related work in this area of research, and the contributions of this thesis. Chapter 3 discusses the Reverse State Reconstruction algorithm, which is used as a non-sampling bias removal technique to accurately attain sampled measurements. Chapter 4 discusses the Single-Pass Sampling Regimen Design methodology, which is used as a sampling bias removal technique in the construction of valid and representative sampling regimens. Chapter 5 discusses fundamental challenges that currently prevent the accurate and reliable sampling of single-application, multi-threaded workloads. Chapter 6 discusses the Barrier-Interval Time-Parallel Simulation methodology, which provides designers

with a highly accurate and fast technique to accelerate the simulation of certain classes of multi-threaded workloads. Chapter 7 concludes and discusses future work.

CHAPTER 2

STATISTICALLY SAMPLED SIMULATION

2.1 Sampling Basics

A population is generally described as a collection of naturally occurring elements (e.g., people, animals, plants, or objects). In statistical theory, a population is defined as the "hypothetical set of all possible observations of the type which is being investigated" [67], and contains all elements from which a sample can be taken. The total number of members in a population is the population size, which may be finite or infinite. Measuring the characteristics of an entire population is often impossible or impractical. For example, it would be impossible to measure the average height of all humans. Architects can measure certain populations, such as running an entire program to completion, but this is often impractical. Exploring the entire design space for the best architectural design is usually impossible. The primary goal of statistics is to characterize and to make inferences about populations given a collected sample. The population from which a sample is taken is called the parent population. Populations are commonly represented by their distribution of values with a probability distribution function (pdf) or cumulative distribution function (cdf).

A sampled distribution is the probability distribution of a given statistic under repeated sampling of the parent population. The individual observations within a sample are called sampling units. Different variables are used to differentiate between estimates based on sampled population and the true parent population characteristics. These

differences are shown in Table 2. There are two key differences between the sampled and true parent characteristics. First, the sampled mean, variance, and standard deviation are called the estimates of the mean, variance, and standard deviation since their true values are unknown (their true values requires measuring all population members). Second, the denominator of the variance equation is different for the sampled distribution, and is called the unbiased sample variance. For large populations, the differences between the biased and unbiased sample variances are minimal. The sampled distribution will more closely resemble the properties of the parent distribution as the sample size increases, assuming the appropriate bias[4] has been reduced. Mathematically, as the sample size increases, $n \rightarrow \infty$, the sample mean will approach the true mean, $\overline{x} \rightarrow \mu$, and the sampled standard deviation will approach the true standard deviation, $s \rightarrow \sigma$[1]. The likelihood of obtaining a good estimate of the mean is inversely related to the sample variance [84]. Populations with higher variance are more difficult to sample (i.e., they are less precise since sample means vary) and require more sampling units than populations with lower variance. Assuming large populations, the appropriate sample size, n, for a mean may also be determined with $n = (z^2 s^2)/\varepsilon^2$, where ε is the acceptable margin of error and z is the appropriate z-score (see below).

[4] The accuracy of all sampling procedures relies upon the reduction of sampling and non-sampling bias. Their definitions and compensatory techniques are described later in this chapter.

Table 2: Descriptive Statistics for Parent and Sampled Distributions

	Parent Distribution	Sampled Distribution
size	N	n
mean	$\mu = \frac{\sum_{i=1}^{N} x_i}{N}$	$\overline{x} = \frac{\sum_{i=1}^{n} x_i}{n}$
variance	$\sigma^2 = E[(X-\mu)^2]$ $\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)$	$s^2 = E[(X-\overline{x})^2]$ $s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})$
standard deviation	$\sigma = \sqrt{\sigma^2}$	$s = \sqrt{s^2}$

The student-t distribution is an important distribution in parametric[5] statistics when estimating the true mean and standard deviation, and depends only upon the degrees of freedom. Examples of the student-t distribution and its relationship with the standard normal distribution are shown in Figure 3. As the degrees of freedom are increased, the student-t distribution converges towards the normal distribution. The standard normal distribution is a normal distribution, where $\mu=0$ and $\sigma^2=1$. In order to fully explain these concepts, one must first understand the concept of degrees of freedom (v). Degrees of freedom refer to the number of independent values that are free to vary. Suppose, for example, a sample size of 5 is taken with the following values: 34, 12, 49, 26, and 73, with a mean of 38.8. If the sample mean is multiplied by five, and individual elements are subtracted one at a time, then the last element remaining will equal the difference ($38.8 \times 5 - 34 - 12 - 49 - 26 = 73$). Given any sample, the deviations between the individual values from the sample mean must sum to zero, $\sum(x - \overline{x}) = 0$. This relationship exactly describes the situation with the student-t distribution. Since the true

[5] Statistical tests and procedures are classified by their underlying assumptions. Parametric statistics make assumptions that sampled data come from certain distributions. Non-parametric statistics make no such assumption, and data may come from any type of distribution (they are also commonly called distribution-free methods).

mean is unknown, it is estimated from the sample mean. Substituting the true mean for the sample mean causes the degrees of freedom to be reduced by one, since one sampled value is no longer free to vary. Thus, given any sample of size *n*, there exist *n*-1 degrees of freedom.

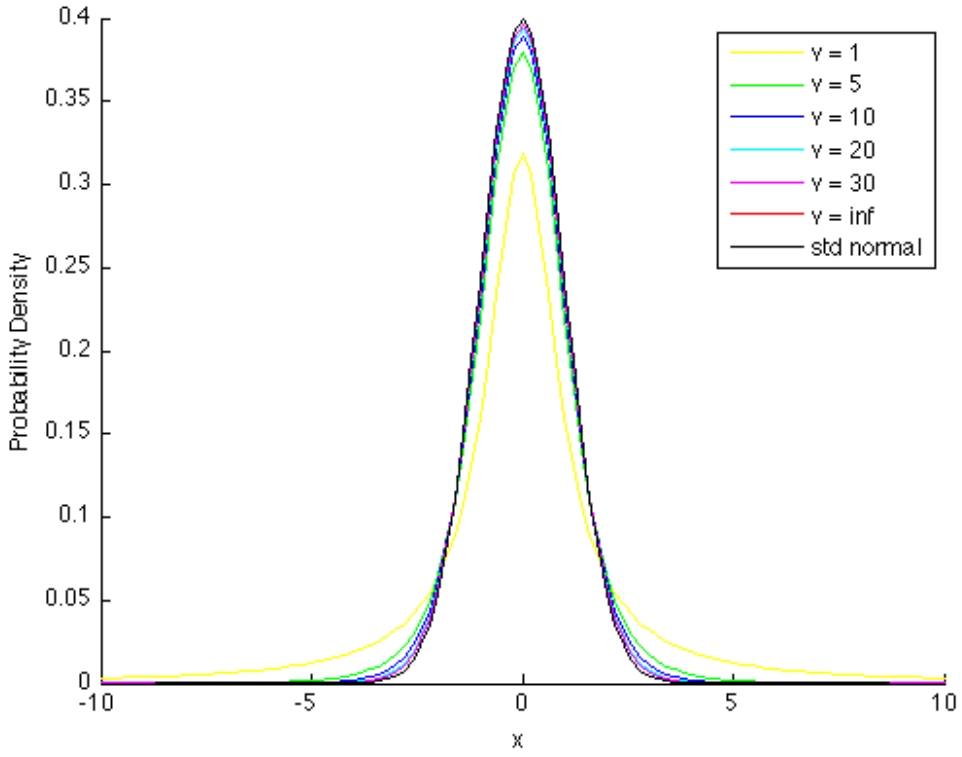


Figure 3: Student-t and the Standard Normal Distribution.

As the degrees of freedom increases, the student-t distribution converges towards the standard normal distribution. Once the degrees of freedom is greater than 30, the student-t and standard normal distribution may be used interchangeably for the calculation of two-tailed tests (two-tailed tests are used by confidence interval

calculations). If it is assumed a sample was taken from a normal distribution, two-tailed tests indicate the range in which an estimate is likely to reside according to some confidence level, α. For a student-t distribution with infinite degrees of freedom, a confidence interval with a 95% confidence level is the points along the x-axis (called z-scores) that contains 95% of the population. In this example, the z-scores occur at -1.96 and +1.96. When referencing a table for these values, the percentages for two-tailed tests commonly indicate the percentage of population elements not contained in the distribution. Z-scores for a 95% confidence level then become $z_{5/2}$ or $z_{0.025}$.

Many statistical tests assume samples are taken from a normal distribution. However, what if that is not the case? In many instances, the Central Limit Theorem (CLT) may be used to justify experiments that obtain samples from non-normal distributions. The CLT states that, for samples taken from a normally distributed population, the sample means will be normally distributed where, $\overline{x} = \mu$ and $s = \sigma/\sqrt{n}$. Furthermore, when sampling populations of any distribution shape, the sample means will be normally distributed whenever the sample size is 30 or more. As *n* increases, the distribution of sample means tends towards normality. The CLT *does not* indicate the distribution itself becomes normal. The parent distribution is immutable towards sampling; it cannot be changed. The CLT, itself, is based upon assumptions that must be verified for a sampling process to be robust towards non-normality. Sampling units (i.e., random variables) must be independently and identically distributed (i.i.d). Sampling units must be drawn from the same parent distribution (they are identical) and they must be uncorrelated (independent). The population mean and variance must also exist and be finite. A standard example of a distribution with a variance that does not exist is that of

the Cauchy distribution. Since the mean and the variance of the Cauchy distribution are not defined, their estimation through sampling will not be successful.

Confidence intervals are an interval estimate used to indicate the reliability of a sample estimate. The width of a confidence interval is inversely proportional to the certainty that should be associated with a sample estimate. For a given confidence level, sample size, n, and standard error, $s_{\overline{x}}$, a confidence interval is calculated as follows.

$$s_{\overline{x}} = \frac{s}{\sqrt{n}}, \qquad \overline{x} \pm z \cdot s_{\overline{x}}$$

The width of the interval is therefore inversely proportional to the sample size, and directly proportional to the standard deviation and z-score. Z-scores for various confidence levels with infinite degrees of freedom are shown in Table 3. Since the student-t sufficiently approximates the normal distribution when the degrees of freedom is greater than or equal to 29, z-scores with infinite degrees of freedom may be used when the sample size is greater than or equal to 30.

Table 3: Confidence Interval two-tail Z-scores with Infinite Degrees of Freedom

Confidence Level	Critical Value
50%	0.674
80%	1.282
90%	1.645
95%	1.960
98%	2.326
99%	2.576
99.5%	2.807

Confidence intervals indicate the range of values that contain the true population mean, at the "probability" specified by the confidence level. If all necessary and sufficient assumptions have been met, then the confidence intervals of random samples

will bracket the true mean at a probability specified by the confidence level. It is expected, under repeated sampling, that sample means will vary for each unique sample. The proportion of sample confidence intervals that correctly contain the true mean is called the coverage probability, and should approximately equal the confidence level. For example, for 100 repeated trials a 95% confidence level should contain the true mean for approximately 95 samples. Conversely, the true mean could be expected to lie outside the calculated confidence interval for 5 samples. A common misinterpretation of the confidence interval is to say that the true population mean is between a particular sample's confidence interval with a 95% probability. Probability statements made regarding individual confidence intervals are not technically correct. The true population mean (although unknown) is a fixed value that is estimated through the sampling process. A particular sample's confidence interval will either contain or not contain the true value. With respect to any individual sample estimate, the true mean is contained within the estimated interval at a probability of either 0% or 100%.

2.2 Sampling Techniques for Microarchitecture Simulation

A wide variety of sampling methodologies exist. The selection of the appropriate sampling strategy is dependent upon a number of factors; however, this discussion will be limited to the methods commonly used in architectural simulation. Commonly used sampling methods include: simple random sampling, systematic sampling, cluster sampling, and stratified sampling. Sampling strategies are also characterized by their selection criteria, which may be random or systematic.

Many statistical tests and procedures assume a uniform distribution for sample selection. The uniform distribution assigns a probability to every individual within a

population, of size *N*, where each individual has an equal probability for selection, 1/*N*. Random processes that follow no deterministic pattern are commonly used to obtain a uniform distribution. Random numbers may be produced through the use of stochastic processes, random number tables, or other mathematical algorithms such as pseudorandom number generators. Although random number generators are fundamentally deterministic if one knows their implementation details, they follow the uniform probability distribution. Since pseudorandom number generators are close approximations of uniformly random processes, they may be accurately used for sample selection. The term probability sampling is used to refer to any sampling method that utilizes random selection. Although random samples may be unrepresentative based upon the "luck of the draw", the accuracy of sample estimates may be inferred either by increasing the sample size or through repeated trials.

Systematic sampling refers to the use of any non-random process for sample selection. The most common form of systematic sampling involves the selection of every k^{th} element of the population. For a population of size, *N*, and a desired sample size, *n*, the sampling rate, 1/*k*, may be calculated, where $k = N/n$. Systematic sampling refers to the use of any deterministic function for sample selection, which may be a simple periodic event or a highly complex algorithm. Under certain conditions, systematic sampling is generally regarded as a sufficient approximation to random sampling. Systematic selection is preferable to random selection when the variance of a systematic sample is greater than the variance of the population. If sufficient knowledge is known regarding the parent distribution, then periodicity may be exploited for simple and efficient sampling strategies. In many instances, the characterization of the parent

population is prohibitively expensive. Since the parent populations (i.e., performance metrics) are dependent upon the workload, workload input, and interactions with the underlying hardware, the characterization necessary to perform these types of optimizations often result in intractable simulation times. The main caveat when considering applying systematic sampling methodologies is they should not be used if the parent population exhibits periodicity. Due to the phase behavior of program executions, periodic behaviors may indeed exist. If observations are systematically placed at the same frequency (or any other higher-order harmonic) as the periodicity, then significant errors may be introduced. Systematic samples obtained from microarchitectural simulations may therefore be unrepresentative.

Simple random sampling (SRS) involves the selection of n elements from a population. Sample selection occurs entirely by chance. Every element in the population has the same probability of being selected, and every element in the population must be available to be selected. Selection may be performed either with replacement or without replacement. When sampling with replacement, elements selected for inclusion into the sample may be selected again (i.e., they are replaced into the population pool from which elements are selected). When sampling without replacement, population elements may be selected only once. Although, strictly speaking, sampling without replacement violates the assumption of independence, the error introduced in sampling large populations is negligible since the probability that any one individual is selected is extremely low. SRS is a simple sampling strategy which requires minimal information about the population of interest. Population elements are the measured processor attributes over individual cycles or instructions. The application of SRS to processor

simulation, however, is infeasible since no meaningful characteristics may be measured through the execution of individual cycles or instructions within the pipeline. Furthermore, the warm-up necessary to collect metrics at this granularity are typically cost-prohibitive. Since SRS is cost-prohibitive for processor simulation, researchers have typically relied on cluster sampling or stratified sampling methods.

Stratified sampling is a sampling process where the entire population is divided into distinct, non-overlapping groups. These groups, called strata, must contain all population elements and each element must only be associated with one stratum. The assignment of population elements to strata is called stratification. Strata should generally contain homogeneous members, since the purpose of stratification is to reduce the variability of the population subgroups. By reducing measured variance, stratified sampling techniques often require smaller sample sizes compared to SRS. Stratification is also beneficial in reducing sampling bias by preventing unrepresentative samples. Each stratum is then sampled either randomly or systematically to obtain strata estimates. Two forms of stratified sampling exist: proportionate and disproportionate stratification. In proportionate stratification, each strata is sampled with a sample size proportional to the size of the population (i.e., all strata are sampled according to the same sampling fraction). This technique serves to emphasize important subpopulations, and generally disregards unimportant ones. Disproportionate stratification allows for the different strata to be sampled at different rates, and allows the precision of certain strata to be maximized. Typically, this involves the sampling sizes for individual stratum to be determined by their observed variability. Larger samples are taken from strata with larger variance, and smaller samples from strata with smaller variance. Sample estimates

may be formed by combining strata estimates using a weighted mean. Stratified sampling is an accurate sampling methodology that can be used to obtain estimates more efficiently than SRS. However, stratified sampling requires detailed, population-specific knowledge to perform the stratification process and is more complex than SRS. For architectural simulation, the population knowledge necessary to perform accurate stratification is often not known *a priori*. The discovery of such knowledge in architectural simulation is a very costly process, often requiring full simulations [56], [57]. Thus, cluster sampling is more commonly utilized to perform sampled simulation than are stratified sampling processes.

Cluster sampling is a sampling process whereby the entire population is divided into groups, or clusters. Similar to stratified sampling, individual population elements must be contained in only one cluster. A number of clusters are selected (systematically or randomly) for inclusion into the sample. Unlike stratified sampling, clusters are determined without requiring detailed knowledge of the population. Clusters may be defined using population element characteristics such as geographical location, or, as in the case of architectural simulation, program location. Two forms of cluster sampling methods include one-stage sampling and two-stage sampling. In one-stage sampling, all members within selected cluster are included into the sample. For two-stage sampling, a sample of clusters is taken, and individual elements within the clusters selected during the first stage are selected for inclusion. The application of two-stage cluster sampling to processor simulation suffers from the same disadvantage as SRS (i.e., measurement of individual population elements do not provide meaningful estimates). Thus, sampled processor simulation is largely dominated by one-stage cluster sampling. It is generally

assumed that clusters are approximately the same size, since cluster sampling requires every element in the population has an equal probability for selection. However, techniques such as probability proportional to size (PPS) sampling may be used if the cluster sizes vary dramatically. PPS selection assigns population elements a selection probability based upon their proportion to the total population size. Since PPS requires detailed knowledge of population members to assign their relative importance, sampled architectural simulation typically involves similarly sized clusters. Cluster members should generally be heterogeneous in nature, since each cluster should represent a small-scale version of the population. It is also beneficial if cluster means are homogenous, since the variance will be lower and the sampled estimates more precise. One advantage of cluster sampling is it can be less costly than other sampling methods. However, for a given sample size, *n*, cluster sampling typically suffers from a loss of precision compared to stratified sampling and SRS. Samples taken using cluster sampling must generally be larger to compensate for this precision loss.

2.3 Sampling Error

The difference between the sampled mean, $\overline{x}$, and the true population mean, μ, is an instance of sampling error, $\overline{x} - \mu$. Efforts must be made to minimize sampling error as much as possible to avoid drawing incorrect conclusions from the distribution being sampled (or distributions, depending upon the statistical test). At first glance, the estimation of sampling error might seem impossible since the true population is unknown and cannot be measured, assuming the population is very large or infinite. How can the error be quantified if its calculation is dependent upon an unknown value? The estimation of sampling error may be performed through standard error, $SE_{\overline{x}} = s/\sqrt{n}$. If

an estimator is unbiased (the estimate of the mean is unbiased), the standard deviation of the sampling error may be estimated from the standard deviation of the sample itself. An estimator is said to be unbiased if it produces a value that is approximately equal to the parameter being estimated. The justification for standard error is the standard deviation of the difference between the expected value of a random variable and the random variable (i.e., the error) is approximately equal to the standard deviation of a random variable. The accurate characterization and minimization of sampling error, however, is dependent upon reducing two types of bias, which may cause sample estimates to significantly differ from their true values. These two types of bias are sampling bias and non-sampling bias.

2.3.1 Sampling Bias

In order for a sample to provide accurate estimates, the elements contained within the sample must be representative of the overall population. If the elements are markedly different from the sampled population, then the sample contains sampling bias. Sampling bias is caused by the over-representation or under-representation of specific population elements, and is strongly related to the method of sample selection. Under any selection criterion, sampling bias is introduced if the chosen elements do not provide an accurate representation of the population. Using random selection, increasing the sample size will reduce sampling bias. If each element has an equal probability for selection, the sampled distribution will contain population elements with approximate proportions as the parent distribution. Elements that occur more frequently will be more likely to be included into the sample, and vice versa. Increasing the sample size for

systematic selection will also reduce sampling bias, assuming no periodicity exists in the population.

The reduction of sampling bias in sampled processor simulation involves the construction of a representative sampling regimen. A sampling regimen defines the number of clusters, the cluster size, and potentially cluster placement. Under random selection, cluster placement is determined automatically. Under systematic selection, cluster placement is dependent upon the sampling rate and the random placement of the first cluster. Both systematic and random selection techniques rely upon the knowledge of the population size, *N*, as the total number of dynamic instructions contained in the workload.

2.3.2 Non-sampling Bias

The formal definition of non-sampling bias is the set of all bias that is not sampling bias. Non-sampling bias refers to any effect or condition that causes sample estimates to deviate from their true values, excluding derivations caused by sample selection. In other words, non-sampling bias is the error of a sample estimate that would be obtained even if the entire population were measured. A common interpretation of non-sampling bias is to view it as measurement error. Unlike sampling bias, increasing the sample size has no effect upon non-sampling bias. Many different factors may be considered non-sampling bias, including: human error, biased survey questions, false information given during a survey, etc.

In sampled processor simulation, non-sampling bias is created by differences in state that cause dissimilar measurement with respect to the full (un-sampled) simulation. For an arbitrarily located cluster, performance measurements are dependent upon the

instructions being executed. The performances of the instructions are dependent upon their interactions with the underlying hardware, which may be affected by the execution of previous instructions. For example, previous memory accesses generate fill requests, from which future requests may be converted to hits. If the effects of prior accesses are not considered, measured performance may differ significantly. The unknown state at the beginning of a cluster's execution is commonly referred to as the cold-start problem [23]. The strategies and algorithms used to approximate the hardware state of the (unknown) full simulation are called warm-up methods; they warm the hardware environment to ensure measurements are representative. Generally, warm-up is necessary for any "stateful" structure; that is, a hardware component that contains a significant amount of information and the failure to approximate its un-sampled state will appreciably impact the sampled measurement. For uniprocessor environments, the two most important components which introduce non-sampling bias are the memory subsystem and branch predictor. Without warming these structures, cluster measurements could be biased due to cache misses and branch mispredictions.

2.4 Statistically Sampled Uniprocessor Simulation

Cluster sampling involves the selection of clusters from the population space for measurement. In processor simulation, clusters are formed through the selection of contiguous groups of instructions from the dynamic instruction stream. The instructions, themselves, are not technically members of the population being estimated. Rather, the population consists of parameter values measured during the simulation of these clusters. Clusters sizes may be defined according to any number of criteria, including: instructions fetched, instructions executed, instructions retired, etc. Any definition is sufficient if its

application results in approximately sized clusters. Selected clusters are simulated to estimate any mean-based attribute desired by the user (e.g., IPC, branch prediction accuracy, cache performance, etc.). Metrics with no distribution, such as the maximum writes observed by a memory location or the total execution time, cannot be estimated by sampling.

This section describes how cluster sampling is applied to uniprocessor simulation. Unfortunately, accelerating a single application multi-threaded workload running on a multi-/manycore architecture through sampling remains an open research problem. The challenges that currently prevent this approach are described in detail in Chapter 5.

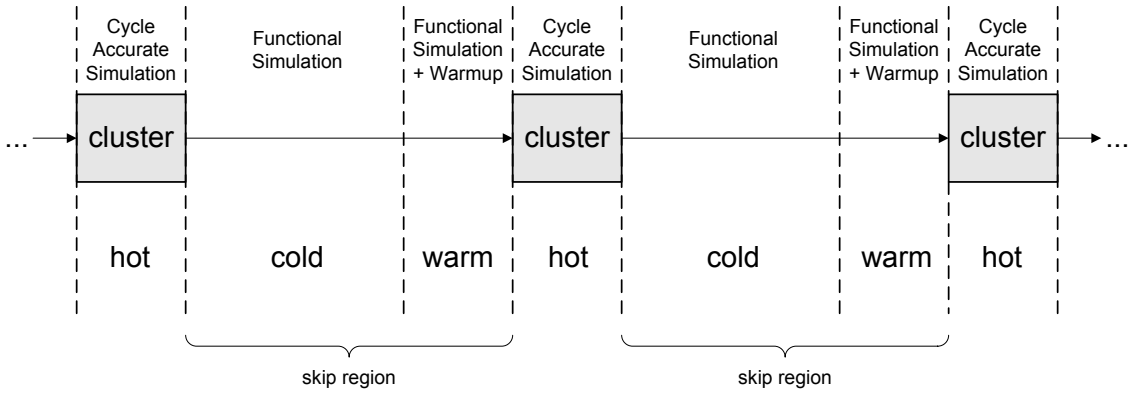


Figure 4: Cluster Sampling Methodology for Sampled Simulation

Figure 4 shows the general cluster sampling process for an execution-driven processor simulation. The horizontal line represents the entire dynamic instruction stream for a particular workload, and consists of three execution phases: hot, cold, and warm. The hot phase is used to obtain cycle-accurate measurements which are included

into the sample. Generally, hot simulation consists of normal cycle-accurate simulation; the complete system (pipeline, functional units, memory hierarchy, branch predictor, etc.) is modeled in full detail. The hot phase can also optionally contain a detailed warming period. After a cluster measurement, the hot phase then transitions to the cold phase. The transition between the hot and cold phases must be performed carefully to ensure previously in-flight instructions have been resolved, so their effects do not contaminate the next cluster measurement. In-flight instructions may be resolved by gating the instruction fetch unit to allow instructions to drain naturally, or by squashing all in-flight instructions.

The cold and warm phases are used to transition simulation to the next measurement location. The instructions skipped between clusters are usually referred to as the skip region or the gap. During the cold phase, functional simulation is performed to ensure correct architectural and functional memory states necessary to skip correct path instructions. No other components are modeled during the cold phase. Simulation then transitions from the cold phase to the warm phase. During the warm phase, functional simulation is performed concurrently with functional warming. Functional warming involves the application of an instruction's effects upon the appropriate component in a functional manner. For example, if a memory reference misses in the cache, the corresponding set is filled with the appropriate tag. Evictions from the set are propagated to lower cache levels, and TLB updates are made as necessary. The time necessary to perform each of these operations, such as fetching the data from main memory, are not modeled. Typically, the components modeled during functional warming include the memory hierarchy and branch predictor. Since functional warming

involves the simulation of a small subset of simulator components at a lower level of detail than in hot execution, warm phase execution is extremely fast.

The simulation framework shown in Figure 4 provides a generic interpretation of cluster sampling. Depending upon the warm-up scheme, the actual simulation flow may differ. For example, the cold phase may be supplanted entirely by the warm phase if functional warming is performed over the entire skip region. Alternatively, the skip region may be performed entirely with cold execution if detailed warming is used for warm-up. Detailed warming relies upon detailed cycle-accurate simulation to warm processor state. Upon the transition to a hot phase, cycle-accurate simulation is performed for a specified number of instructions (or cycles). Without the warm phase, it is expected that cluster measurements would initially be heavily biased. As more instructions are simulated in full detail, non-sampling bias is reduced. To avoid the initially high bias from affecting the cluster measurement, system statistics are reset after the detailed warming period is complete, and measurements are based on the execution of the remainder of the cluster.

Sampling involves a tradeoff between speed and accuracy. Larger samples are more likely to provide accurate estimates, but at the expense of longer simulation times. Smaller samples may have extremely short runtimes, but provide inaccurate estimates. Thus, appropriate sampling regimens must be designed carefully.

2.4.1 Trace-driven vs. Execution-driven Sampling

Warm-up methods have been proposed for both trace-driven and execution-driven environments. To reduce storage costs, skip region instructions are typically omitted from sampled traces. As a result, the instructions within the gap are generally

unavailable for warm-up in sampled trace-driven environments. Techniques have been proposed to warm processor state using only the cluster instructions for warm-up [25], [28], however, the exclusion of gap information generally results in estimates with greater error. In execution-driven environments, the functional data produced by gap instructions are available from the functional simulator. Warm-up methods that perform functional warming over the entire skip region (e.g., SMARTS [80], [81], [82], [84]) are typically more accurate. However, the high accuracy of full functional warming has its cost. Although warm phase execution is extremely fast compared to hot execution, the time required to perform functional warming over large program portions tends to dominate sampled simulation times. Although simulation is slowest during the hot execution phase, very few instructions (as a percentage of the workload) are executed in full detail. For many warm-up techniques of execution-driven simulation, the times required to skip between clusters consumes the majority of sampled runtimes. One advantage of sampled trace-driven simulation is that the times required to skip between clusters is minimal. Simply reading the appropriate trace element for the next cluster performs the progression to a subsequent cluster.

2.4.2 Checkpoints vs. Functional Fast-Forwarding

The simulation of gap instructions via warm or cold execution typically dominates the runtimes of sampled simulations, and therefore limits speedup. Rather than incurring this cost for each simulation, the necessary state may be saved to disk and reused during future simulations. Necessary state is typically comprised of the architectural and functional memory state, as well as the functional state of any component that requires warming (i.e., the branch predictor and cache hierarchy). The

files containing these data are called checkpoints. Instead of skipping vast portions of the workload for each simulation, previously generated checkpoints provide a faster alternative. The use of checkpoints has facilitated dramatic reductions in sampled simulation runtimes, [77], [80], [81], [82]. For example, sampled simulation runtimes can be reduced from hours to minutes [82]. Checkpoints may also be leveraged for even faster simulation since each cluster may be simulated in parallel. The accuracy of any "checkpointed" state is dependent upon the warming method used to initially generate the checkpoints.

Although checkpoints provide a fast alternative to skip region processing, they suffer several important disadvantages. One major problem with checkpoints is their storage cost. For example, 36 TB of disk space is required to store the checkpoints necessary to simulate the SPEC2K suite with 10,000 clusters per workload [81]. Although checkpoint sizes may be lowered through compression techniques, their storage remains high. Optimizations to reduce checkpoint storage costs have been proposed. Wenisch, et al. [81] dramatically reduced the cost of checkpoints through a post-processing filter to produce live-state checkpoints. Live-state checkpoints store data accessed in the cluster, and reduced storage requirements from 36 TB to 12 GB for 1MB caches. Checkpoints are required for each cluster, and are strongly tied to each cluster's location. Increasing the sample size increases the number of checkpoints and their associated storage. If a sample is taken with the common practice of systematic sampling according to a sampling rate, increasing the sample size will change all cluster locations and require the generation of new checkpoints. Similarly, taking a different random sample (i.e., a unique random seed) will have the same effect. Sampled simulations that

perform skip region processing with functional fast-forwarding are much slower than those utilizing checkpoints, but incur no storage overheads.

Checkpoints are also not robust to certain microarchitectural changes. If significant architectural changes are made to the warmed components, previously generated checkpoints may no longer be applicable. Checkpoints could also be enhanced if the functional state of multiple microarchitectural configurations were provided, but would incur increased storage.

2.5 Warm-up Methods

Many algorithms have been proposed to reconstruct the architectural state in warm execution, and have been the main focus of many sampling studies. Many algorithms to perform warm-up have been proposed, including: BLRL [29], MSE [33], MRRL [34], and SMARTS [84].

The sampling of workloads was originally applied to cache simulations [22], [42], [49] and then later extended to processor simulation [23]. Since that time, a number of sampling techniques have been applied to hardware simulation, which include cluster sampling [12], [13], [21], [22], [23], [50], [71], set sampling [30], [42], [50], and stratified sampling [61]. Although these methods differ in their sampling and non-sampling bias reduction techniques, they all use sampling to reduce simulation times. A brief survey of the various sampling strategies that have been developed is presented to provide the necessary background to effectively explain and demonstrate the contributions of this thesis.

2.5.1 Cache Warm-up Techniques

2.5.1.1 Excluding Unknown References

Laha, et al. [49] investigated warm-up methods for cache designs utilizing a sampled trace-driven simulation environment to estimate cache miss rates. Clusters were placed immediately following context switches, and caches at the beginning of each measurement were empty. Experiments showed set-associative and direct-mapped caches smaller than 16KB did not require warm-up for accurate estimates. For smaller caches, empty tag-stores immediately following a context switch were sufficient since other running processes served to overwrite cache data. In this case, warm-up was achieved through the selective placement of cluster measurements rather than a functional warming technique. The distributions of the sampled and un-sampled miss rates were compared using the Kolmogorov-Smirnov (KS) two-sample test. The KS-test is a non-parametric and distribution free test used to determine if there was sufficient evidence to reject the null hypothesis; in other words, the KS-test was used to demonstrate that sampled measurements were representative of the un-sampled population. The authors observed that miss rates were not normally distributed, and required approximately 35 measurements per sample (more than the 30 required by the central limit theorem). Using cluster sizes of 5k, 10k, and 20k references, miss rate errors were 5.5%, on average.

Although context switches approximated the state of small caches, they could not be exploited to accurately reconstruct larger designs. Since caches larger than 16 KB retained significant amounts of data across context switches, warm-up was extended to include *primed sets*. At the beginning of a cluster measurement, cache miss rates were

high since the cache was empty. As cache lines filled with blocks, the miss rates were reduced. After a cache line was filled with unique references, it was considered primed and then counted towards miss/hit-rate statistics (thereby excluding the affects of unknown references). The authors investigated various warm-up strategies based upon set priming, and concluded upon a T_2-T_5 priming range for statistics collection. T_2 was used to refer to the first non-MRU reference to a primed set, and T_5 was used to refer to the end of a cluster measurement. Measurements of a primed set immediately following priming to the end of a cluster typically underestimated the miss ratios due to a high number of MRU hits. Since the last fill was excluded from miss rate calculated, the inclusion of the following MRU hits caused the miss rate to be underestimated. The authors chose to exclude these references as a corrective factor.

The techniques used by Laha, et al. [49] were effective in reducing non-sampling bias. However, the systematic placement of clusters immediately following context switches may introduce sampling bias that could negatively impact estimate errors for other workloads.

2.5.1.2 Estimation of Unknown Fill References

Wood, et al. [83] also performed sampled simulation utilizing a trace-driven environment for set-associative and direct-mapped caches, but performed warm-up differently than in [49]. Rather than excluding measurements of unknown references, this approach developed an accurate estimator of the distribution of unknown references. The sampled miss rate of a cache with unknown state at the beginning of a cluster may be calculated as,

$$m = \frac{M + \mu U}{R}$$

where m is the miss rate, M is the total count of known misses after reconstruction, μ is the unknown reference miss rate, U is the unknown reference count, and R is the total reference count. The exclusion of unknown references U from the sampled miss rate calculation m implicitly assumed miss rates for the unknown references were the same as the sampled measurement. Wood, et al. [83] demonstrated the unknown miss rate was much higher than the cluster miss rate, and developed an estimator based upon renewal-reward theory.

Previous studies commonly approximated μ, and assumed unknown references hit (hot-start, where $\mu = 1$) or missed (cold-start, where $\mu = 0$). If $M >> U$, then miss rate accuracy m was generally acceptable. However, for large caches, this condition was typically not satisfied and resulted in overly pessimistic estimates for cold-start, and overly optimistic estimates for hot-start. Although inaccurate for larger cache designs, cold-start and hot-start estimates may be used to obtain lower and upper bounds for the unknown reference miss rate (the true unknown miss rate must lie in between these bounds). Other studies compensated for unknown miss rates by using large clusters, since large values of M amortized error and minimized the differences between cold-start and hot-start approximations. Cache priming is an alternate technique, and involved the reconstruction of the entire cache, or individual cache sets, before collecting detailed statistics. If entire cache priming was performed, then low miss rates required clusters to be extremely large before measurement could occur. The priming of individual cache sets was better in this regard, since detailed measurement did not require clusters to be as large. However, the priming of individual cache sets assumed similar distributions for μ and m, which was shown to be false.

A predictor of the unknown reference miss rate was based upon renewal theory, where individual cache blocks were modeled as generations that transitioned between alive and dead states. Each generation was modeled through,

$$G_j = L_j + D_j$$

where G_j is the cache block's generation (the j^{th} miss of a particular block), L_j is the live time, and D_j is the dead time. A block was alive if a future access resulted in a hit, and considered dead if a future access resulted in a miss. A miss event for a particular block resulted in a new generation (G_{j+1}) of that block. By modeling cache blocks as either alive or dead, renewal theory allowed the authors to estimate the miss rates for arbitrary cache blocks at time *t* through,

$$P\{Block_t \; is \; a \; miss\} = \frac{E\left[D_j\right]}{E\left[G_j\right]} = E[\mu]$$

where E[D_j] and E[G_j] are the expected values of random variables D_j and G_j, and E[μ] is the estimator of the unknown reference miss rate. During sampled simulation, cache configurations were completely warmed (entire cache priming) before detailed measurement began, followed by detailed measurement of 5M references. Sample sizes in this study ranged between 19 and 35 clusters. After cache priming, m and μ were measured and compared against the estimates calculated through the average block lifetimes. This (long) model assumed every block within the cache was accessed at least once within the cluster. Experiments conducted in this study demonstrated E[μ] was an unbiased estimator of the unknown reference miss rate, where the majority of workloads exhibited less than 10% relative error. Although one workload exceeded a relative error of 20%, it should be noted this study estimated the cache miss rate (which are generally small values). The calculation of relative error on small values tends to bias results

towards higher error values, since the magnitude in percentage difference between the exact value and the approximate value will generally be larger.

A short model was also developed to estimate unknown reference miss rates in cases where every cache block was not referenced at least once. However, the derived (short) model had much higher error rates than those observed for the long model.

Fu and Patel [30] proposed an alternate technique to estimate unknown fill references based upon miss distance, *d*. The miss distance was defined as the number of memory references that occurred between cache misses, including the first miss. As in [83], the authors observed the unknown fill miss ratio were much higher than the overall miss ratio (fill references are not random trace references, and have much different behaviors than known references). However, this study differed significantly from previous works since it predicted the distribution of miss distances rather than miss ratios. Although the miss ratio is a commonly used performance metric for caches, it is inadequate when other system components are modeled or if cache misses can overlap (e.g., MSHRs) [30]. Models that solely estimate the unknown reference miss rate, μ, are inadequate when other components are included, since detailed simulation requires these references be classified as a hit or miss. Thus, the authors advocated the use of miss distances as a more detailed measurement of cache performance (the reciprocal of the average miss distance, 1/*d*, is the miss ratio).

In this scheme, samples of 40 clusters were placed randomly throughout the trace, and each cluster contained 200,000 references. At the beginning of each cluster, caches were assumed to be empty. The first access to a unique location within a cluster was an unknown fill reference, as above. Subsequent accesses to same location were

then significant (primed). Unlike previous techniques, cache priming occurred over individual cache blocks rather than complete cache sets or even the entire cache. Clusters measurements were gathered utilizing two phases: a priming phase and an evaluation phase. The priming phase was a form of detailed warming used to reduce the number of unknown fill references encountered during the evaluation phase. Measurements were collected over the evaluation phase, which consisted of detailed cache simulation. Unknown fill references that occurred within the evaluation phase were predicted as hits or misses, and included as part of the sampled measurements.

Unknown fill references were predicted in the evaluation phase using information contained within a miss distance table and cache contents. The miss distance table contained a record of the three most recent miss distances (tables of greater sizes had negligible impact on prediction accuracy). Priming initially began with an empty miss distance table and cache. Fill references were predicted during evaluation as follows: 1) if the history table was empty, the fill was predicted as a hit; 2) if the distance, d, was within the ranges contained in the history table, the fill was predicted as a miss; 3) if adjacent sets (to the set being filled) contained cache blocks of adjacent memory addresses, the fill was predicted a hit; and 4) if all previous conditions were not satisfied, the fill was predicted a miss.

Prediction accuracy was evaluated by comparing miss distances of the sampled and full simulations. Three schemes were evaluated: sampling without prediction, sampling with prediction, and sampling with random prediction. In sampling without prediction, only known (significant) references were used to estimate the mean miss distance. Sampling with prediction utilized the prediction mechanism described above.

Sampling with random prediction randomly predicted fill references as a miss or hit. Of these three schemes, sampling with random prediction resulted in very large error and was the least accurate, sampling without prediction performed adequately, and sampling with prediction performed the best. Sampling with prediction accurately estimated the mean and the standard deviation of the miss distances for a number of different workloads. Furthermore, analysis of the prediction scheme over a number of different cache sizes demonstrated predictions based upon the history table dominated fill predictions for smaller caches, and predictions based upon cache contents dominated fill predictions for larger caches.

2.5.1.3 Set Sampling

Liu, et al. [57] utilized trace-sampling to estimate miss ratios for set-associative caches, but applied sampling to individual cache sets. Individual cache sets were selected for inclusion into the sample and miss ratios were computed exclusively using these sets. In lieu of evaluating the entire cache for a contiguous group of references (as in cluster sampling), set sampling simulated all references that mapped to the selected sets and ignored all others. Set sampling, therefore, extracted population elements based upon spatial locality rather than time (temporal locality). Since chosen sets were simulated over the entire trace, no warm-up was required for detailed measurements and the (non-sampling bias) unknown references problem was avoided.

The random selection of sets is a form of simple random sampling. Under simple random sampling (SRS), each set has an equal probability of being selected for inclusion in the sample. SRS is a simple sampling scheme that works well when the necessary and sufficient assumptions are met (e.g., i.i.d.). This study showed set

sampling with SRS could lead to poor estimates (i.e., low coverage probability), since the population of cache sets violated SRS assumptions. The authors demonstrated that cache sets exhibited high variance and the miss rates of particular cache sets were highly correlated. For example, spatial locality caused adjacent sets to exhibit similar miss behaviors. SRS samples (without replacement) were repeated 500 times for each sample size. Sample sizes ranged between 6.25% (1/16) and 50% (1/2) of the total cache-set population. The authors demonstrated set sampling with SRS was statistically unreliable for small sample sizes. For a sample size of 6.25%, approximately 60% of samples were greater than the 5% margin of error targeted by a 90% confidence level. Even when 50% of the population was sampled, error rates barely reached the target confidence level. Therefore, in order for SRS to be effective for cache sets, a very large percentage of the population must be included within the sample. Set sampling with SRS also suffers from other disadvantages. If sets are selected randomly in each simulation, then the entire trace must be stored[6] to ensure the appropriate references are available. Random set sampling is also problematic in the simulation of multi-level cache hierarchies (e.g., it is unclear how to simulate a cache hierarchy if sets are selected at random) [42].

The authors investigated uniform sampling approaches to determine viable alternatives to SRS. Uniform sampling involved sorting the entire population of cache sets based upon some criterion metric. Using the sorted list, the population was divided into a number of groups. Samples were constructed by picking a cache set from each group. The authors evaluated uniform sampling using four criteria metrics: references,

---

[6] Rather than storing complete traces, traces may alternatively be collected in real time as required by a simulation environment. The dynamic creation of traces during a simulation may require significant overheads.

misses, miss ratios, and weighted misses. The references criteria sorted cache sets based upon the total number of references. The miss criteria sorted cache sets based upon the highest number of misses. The miss ratio criteria sorted cache sets based upon the number of misses divided by the number of references. The weighted miss criteria sorts cache sets based upon the following,

$$W_J = \sum_{j \in J} W_j = \sum_{j \in J} m_j - M_I \times \sum_{j \in J} r_j$$

for a cache containing *I* sets, where J ⊆ I is the sample, M_I is the overall cache miss ratio, m is the sampled misses, and r is the sampled references. The weighted miss measured how sampled sets contribute to miss rate errors (the differences between the sampled miss ratio and the overall miss ratio). Although the selection of sets by misses, references, and miss ratios all performed poorly, the selection of sets by weighted misses had low error rates (less than 1%).

The sampling methodology employed by Liu, et al. [57] was a systematic form of stratified sampling. The authors achieved extremely low error rates by constructing samples based upon the proposed weighted miss scheme. However, the use of a weighted miss characterization required a high degree of population knowledge. In order to determine which sets should be included in the sample, all sets must be simulated across the entire trace (to obtain the overall miss ratio, and the per-set miss statistics used for ranking). This classification was presented as an up-front cost, which could be applied to different cache configurations. As shown through various SPEC v1.0 simulations, the applicability of systematic samples to alternative cache configurations resulted in unpredictable error rates. Furthermore, since sets could not be selected until

the entire cache was simulated for the entire trace, the entire trace had to be kept (and does not reduce the storage costs of collected traces).

Kessler, et al. [42] proposed a *constant-bits* approach for the systematic selection of sets. The constant-bits method filtered trace elements based upon memory references rather than sets. Within each memory reference a subset of bits were checked for a particular pattern (e.g., references with the bit-string constant 0000 in address bit range 11-8). If the reference contained the desired pattern, it was included in the filtered trace for simulation. In this study, the address bits selected for filtering were located above the block offset, and contained a number of index bits. In order for the filtered trace to be simulated for all desired cache configurations the bit range was tailored based upon the maximum block size, the cache size and cache associativity, as well as the desired percentage of selected references. For example, suppose the previous bit pattern of 0000 was selected between bits 11 and 8. The maximum block size of all simulated caches had to be less than 256 bytes (2^8). All cache sizes under test divided by their associativity must also exceed 2 KB (2^{11}). Assuming an equal distribution of all bit-strings in bit positions 11-8, then the pattern 0000 will select approximately 1/16-th of the references (since the other 15 bit patterns are excluded).

The constant-bits method was evaluated over a number of uniprogrammed and multi-programmed workloads. For each simulation, success was based upon the 10% sampling goal. A simulation passed the 10% sampling goal if the estimate error was less than 10%, it used less than 10% of the trace for simulation, and had a coverage probability of 90%. Coverage probabilities were assessed through repeated construction of 90% confidence intervals. The tested multi-programmed workloads generally passed

the 10% sampling goal, however, the uniprogrammed workloads suffered from low coverage probabilities. Confidence intervals generally assume that the estimate of the mean is normally distributed. Due to the central limit theorem, this assumption is typically robust to non-normality unless the underlying population is heavily skewed (i.e., the distribution has fat tails such as in Cauchy distribution). Although mean estimates were normally distributed for multi-programmed workloads, uniprogrammed workloads exhibited several "hot sets" that significantly skewed their distributions. Thus, confidence intervals for these workloads were unreliable, and resulted in low coverage probabilities.

Unlike previous approaches, constant-bits allowed the same filtered trace to be applied to all cache configurations (provided they include the constant bits), and can be used to simulate multi-level caches. Unfortunately, estimates obtained from constant-bits may perform better or worse than SRS, depending upon the periodicity of the cache behavior and the variance captured by the sampled sets [42]. Ultimately, although a few rudimentary guidelines were given regarding bit-range and pattern selection, the criteria used for filtering were arbitrary.

Set sampling has been used to accurately estimate cache metrics such as the miss rate and misses per instruction. However, set sampling has its limitations. Set sampling may result in inaccurate estimates if there are significant interactions between sets, or if outliers heavily skew the distribution. Significant set interactions commonly occur through spatial locality (causing adjacent sets to behave similarly), or when components such as prefetchers, write-buffers, lockup-free accesses (MSHRs), or victim caches are modeled [42].

2.5.2 Processor Warm-up Techniques

2.5.2.1 Sub-component Simulation

Lauterbach [50] sampled processor simulation to estimate CPI performance, but did not apply sampling to the caches. Rather, a separate cache simulation was used to provide state information at the beginning of each cluster. By simulating caches throughout the trace, the non-sampling bias associated with cache state was removed. In this study, 100 clusters of 100,000 instructions were randomly placed throughout the trace. A single-pass cache simulator was used to quickly evaluate many different cache configurations. Cache contents were saved for each cluster location, and used during sampling (i.e., cache specific checkpoints were created). Unknown cache state typically requires longer clusters to compensate for unknown fill references, and can limit speedup over the full simulation. Since the complete cache contents were known, accurate measurements could be obtained through small clusters, resulting in greater speedup. However, the sampling process remained limited by the speed of the cache simulator (approximately 50 kIPS) [50]. The authors indicated the tested SPEC92 workloads contained over 100 billion instructions. Assuming a constant cache simulator speed of 50 kIPS, simulation of the entire SPEC92 suite would require approximately 556 hours, or 23 days. Although the SPEC92 suite contained 20 workloads (6 integer and 14 floating point), most benchmarks only required one day for sampling.

Samples were constructed using a sampling-verification-resampling approach. Sample estimates were verified against instruction frequencies, basic block densities, and cache metrics which tested for sample representativeness. If estimates were not representative according to a specified criterion, then more clusters were added until

measurements converged to their true values. Although effective in producing representative and accurate estimates, this type of sample construction required a full simulation of the entire trace.

Over the tested workloads, less than 0.5% of the instructions were sampled with an overall error rate of less than 1%. Furthermore, 99% confidence intervals were constructed with a margin of error of 2% that contained the true CPI. Over sequential simulation, average speedups of 200x were obtained. Additional speedups (proportional to the number of processors) were also obtained through the execution of individual clusters in parallel.

2.5.2.2 State-reduction Method

Conte, et al. [23] proposed the state-reduction method. Using this technique, performance estimates were evaluated without the full simulation for error comparisons. The simulation environment targeted a Tomasulo-based RISC processor design, which utilized infinite cache and a highly speculative branch predictor. Since an infinite cache was assumed, and no warm-up was required for the cache, warm-up was restricted towards the branch predictor. An analysis of branch behaviors for SPEC92 workloads was performed, which showed many branches occupied significant portions of the execution. Three different warm-up methods were also investigated in the reconstruction of branch predictor state: *fresh-BHB*, *stale-BHB*, and *fixed*. Fresh-BHB began each cluster by clearing all branch predictor state. Due to long branch lifetimes, this warm-up had the highest non-sampling bias, and resulted in the highest error. Stale-BHB began each cluster using stale state from previously executed clusters (i.e., the branch prediction state was never cleared). Although stale BHB state performed much better, significant

errors remained. The fixed warm-up scheme was the best. Fixed warm-up used stale branch predictor state and an additional detailed warming period. Using fixed warm-up, all workloads had less than 3% error and passed 95% confidence intervals (except one workload, where the true IPC was less than 0.01% outside the predicted interval).

The proposed state-reduction method involved the following steps. Non-sampling bias was first removed through the selection of a warm-up scheme, cluster size, and detailed warming period. Next, sampling bias was reduced through the iterative construction of a valid sampling regimen. After specifying a number of clusters, a sampled simulation was performed to calculate an estimate of standard error. The reliance upon standard error to assess sample accuracy is a commonly used statistical technique, which obviates the need for a full simulation comparison. This was obviously desirable, since otherwise sampling could not provide any clear benefit in terms of runtime. Given a maximum standard error threshold, additional clusters were added to until a sufficient standard error for the sample was obtained.

2.5.2.3 SMARTS

Wunderlich, et al. [84], [85] proposed SMARTS warm-up, which used a combination of functional warming and detailed warming to achieve highly accurate sample estimates. Sampling units were systematically placed throughout the instruction stream based upon the benchmark size and the desired number of clusters. Between clusters, all branch and memory instructions were functionally applied to the branch predictor and memory subsystem components through functional warming. Simulation also incorporated a detailed warming for a specified number of instructions prior to the start of the cluster. Detailed warming was used to further reduce bias that could not be

removed by functional warming alone. Since workload behaviors, and therefore performance metrics, are decidedly periodic in nature the authors measured the intraclass correlation coefficient (ICC). Using ICC values, it was demonstrated the tested SPEC2K workloads were not periodic at the selected sampling rates.

In SMARTS simulation, cluster and detailed warming lengths were specified by the user. To help guide these decisions, an analysis was performed to determine appropriate sizes that achieved the lowest error and highest speedup. To determine optimal cluster sizes, the variability of CPI measurements for each workload was calculated as the cluster size increased[7]. For cluster sizes larger than 1k instructions, measured variability did not significantly decrease. The likelihood of a good estimate increased as the measured variability decreased. Therefore, increasing the cluster size resulted in lost speedup opportunities without increasing estimate reliability. Increasing cluster sizes decreased variability since larger cluster sizes had a smoothing effect, as short-term perturbations were averaged across the entire cluster. The amount of detailed warming necessary to completely warm state varied drastically depending upon benchmarks and inputs. Without functional warming, some benchmarks required less than 50k instructions, while others exhibited unacceptable bias even after 500k. With functional warming, the suggested detailed warming size was determined to be 4k instructions for the largest architectural configuration (a 16-wide superscalar design). The selection of the number of clusters was performed using a similar philosophy as in

[7] The authors actually measured the coefficient of variation (CV), which is a ratio of the standard deviation over the absolute value of the mean. The standard deviation, variance, standard error, and CV are all measures of population dispersion.

[23]; a sample profile was performed with 10k clusters to measure variability and the sample size was increased accordingly.

The SMARTS framework is a highly accurate sampling methodology incorporating both functional and detailed warming. For each simulation, the CPI and EPI (energy per instruction) were estimated. Of the 45 tested benchmark and machine configurations, all estimates had less than 2% error and only 6 had more than 1% error. Error rates were dominated by non-sampling bias associated with functional warming. Although functional warming is extremely accurate, it cannot approximate state where exact detailed execution must be known[8] (e.g., out-of-order execution effects and wrong-path execution). Fortunately, out-of-order effects and wrong-path execution do not significantly affect CPI and other performance metrics [15], [84]. Relative to the full simulation, sampled speedups ranged between 35x and 60x. The SMARTS methodology was also compared against SimPoint [73]; in all tested workloads, sampled simulation exhibited lower error and faster speedup relative to the full simulation.

The use of full functional warming for gap processing is one of the most accurate warm-up methods, but is also very expensive [34]. Using SMARTS warm-up, sampled simulations are dominated by functional warming (i.e., 99% of the time is spent in functional warming [82]). Since so few instructions are simulated in full detail, the speed of detailed simulation is largely moot. Faster speedups could be obtained if optimizations were focused upon the functional warming and fast-forwarding operations.

---

[8] This limitation of functional warming has broad and profound consequences for the simulation of multithreaded workloads. The implications upon multithreaded sampling approaches are discussed in Chapter 5.

One such optimization, called TurboSMARTS [82], incorporated reusable checkpoints to provide faster gap region processing.

2.5.2.4 Minimal Subset Evaluation (MSE)

Haskins and Skadron [33] presented a method to determine the minimum required warm-up length to reconstruct cache state. Profiling was performed with a fast cache simulator (sim-cache) on each workload-input pair to characterize unique accesses. MSE is based upon the observation that, given sufficient unique accesses (memory references that do not access the same address), any arbitrary cache configuration will eventually be completely filled. Profiling was performed using two fully-associative caches to track instruction and data references. Cache blocks were tagged with timestamped counters indicating the total number of instructions executed. At the beginning of a cluster, timestamps were used to sort cache blocks in descending order. From the sorted list, the m-th element indicated the number of instructions prior to the cluster that must be simulated to obtain *m* unique accesses. From the profiled information, cache specific MSE equations of the form,

$$t = MSE(N, a, p),$$

were solved to determine a warm-up length, *t*, given the number of cache sets, *N*, cache associativity, *a*, and a user-specified probability of accuracy, *p*. The equations derived for direct-mapped and set-associative caches are shown in Figure 5. Unfortunately, since closed-form solutions to these equations were not known, the determination of *t* was based upon repeated trials with different values of *N*, *a*, and *p*. The authors indicated the longest calculation for the direct-mapped equation took approximately 30 minutes. The calculation times of the more computationally intensive

set-associative equation were not discussed. The sampled simulation may be performed where functional cache warming is initiated *t-instructions* prior to the cluster beginning.

Direct-mapped equation	$p = 1 - \frac{\sum_{k=1}^{\lceil N \rceil - 1} \binom{N}{k} k^m}{\sum_{k=1}^{\lceil N \rceil} \binom{N}{k} k^m}$
Set-associative equation	$p = \frac{\sum \left[\binom{m}{x_1, x_2, \cdots, x_{N-1}} \middle\vert \; s.t. \; at \; least \; \lceil N \rceil \; x_j \geq \lceil a \rceil \right]}{\sum \binom{m}{x_1, x_2, \cdots, x_{N-1}}}$

Figure 5: MSE Cache Specific Warm-up Calculations

This technique is based on the assumption that unique accesses are randomly distributed. For a number of SPEC95 and SPEC2000 workloads, the uniform and unique access distributions were compared using a χ^2 (chi-squared) test. Interestingly, although caches commonly exhibited "hot spot" behavior due to locality, this study showed that the distribution of unique accesses was approximately uniform.

Sampled IPC estimates calculated with MSE warm-up were more accurate than detailed warming. Although detailed warming generally produced accurate estimates (3.74%, on average), a few workloads exhibited high error (16.4%, maximum). Detailed warming experiments utilized 7k instructions prior to each cluster to warm cache state [23]. Since warm-up lengths required by MSE were longer than detailed warming, MSE simulations had lower speedup over the full simulation. Higher speedups were obtained by lowering the confidence level from 99.9% to 95%. Using the lower probability of

accuracy, similar performance measurements were obtained with shorter warm-up lengths (and higher speedup).

The MSE cache warm-up technique is a profiling-based approach to calculate the minimum warm-up length necessary to prime a percentage of the cache. A 100% probability of accuracy is equivalent to complete cache priming prior to measurement. Estimates using MSE warm-up were extremely accurate (0.3% error, on average), but required costly gap profiling. Each cluster position required profiling for each benchmark and input, which was repeated for each random sample. The MSE technique is also specific towards the calculation of warm-up lengths for the first level instruction and data caches [34].

2.5.2.5 Memory Reference Reuse Latency (MRRL)

Haskins and Skadron [34], [35] proposed an approximation of full functional warming, called memory reference reuse latency (MRRL). Although full functional warming is extremely accurate, it conservatively assumes all branch and memory reference instruction effects are necessary to reduce non-sampling bias. This assumption ensures accurate measurements, but does so at the expense of sampled runtimes. Due to temporal locality, the effects of memory references immediately prior to a cluster are more likely to be important during cluster execution. In other words, not all of the effects of branch and memory reference instructions may be necessary to obtain highly accurate cluster measurements. By selectively omitting certain gap instructions, runtimes may be reduced without significantly affecting sample estimates. The goal of MRRL is to reduce warm-up overheads by simultaneously maximizing cold execution phases and minimizing warm execution phases.

Similar to MSE, MRRL is performed via the detailed profiling of each skip region and cluster location. Profiling was performed using three associative arrays (i.e. hash tables) for the instruction, data, and branch reference streams. The purpose of each array was to construct histograms of reuse distances for each of the three types of references for each skip region and cluster pair. Reuse distances were defined as the number of instructions between subsequent accesses to the same address. The histograms were output after the skip region and cluster were profiled, and used to calculate the number of pre-cluster instructions necessary to warm-up a specified percentage of the collected reuse latencies. The warm-up length was also robust towards different branch predictor or cache configurations since their details are not required for its calculation.

The MRRL technique obtained highly accurate sampled estimates which closely approximated full warming. Samples consisted of 50 randomly selected clusters and a cluster size of 1M instructions to estimate the IPC of processor simulations. The user-specified reconstruction percentage chosen was 99.9%. Samples estimates were taken from a number of SPEC2K workloads to compare the runtimes and error rates of MRRL, SMARTS [84], and *nowarmup*[9]. *Nowarmup* performs no functional warming in the skip region. The caches and branch predictor only contain state produced during cluster execution. *Nowarmup* state is persistent across clusters, and is similar to *stale-BHB* warm-up [23]. Although *nowarmup* was expected to contain higher estimate errors, it provided an upper bound for the maximum attainable speedup. SMARTS experiments

[9] *Nowarmup* typically refers to a warm-up policy incorporating no functional warming during gap processing, as well as empty cache and branch predictor structures. In this study, the definition of *nowarmup* is actually more consistent with *stitch* or *stale* warm-up policies.

provided an upper bound on accuracy. Estimates provided through MRRL approached the accuracy of SMARTS and the runtimes of *nowarmup*. SMARTS and MRRL estimates differed by less than 1%, on average. The runtimes of MRRL samples obtained 90.62% of the (fastest) *nowarmup* runtimes.

2.5.2.6 Boundary Line Reuse Latency (BLRL)

Eeckhout, et al. [29] proposed an optimization of MRRL called boundary line reuse latency (BLRL). One disadvantage of MRRL is the calculated warm-up lengths may be too short or too long for a specified level of accuracy. MRRL measures reuse distances over the skip region and cluster, and assumes the reuse distances in both regions are similar. If reuse distances are shorter in the gap than in the cluster, the calculated warm-up will be too short for reconstruction, potentially causing cluster measurements to have higher error. If reuse distances are longer in the gap than in cluster, the calculated warm-up will be longer than what is required to obtain the specified level of accuracy. The goal of BLRL is to approximate MRRL (and therefore transitively approximate full functional warming), to obtain even faster warm-up.

BLRL, like MRRL, utilized reuse distance profiles constructed from skip region and cluster profiling. However, there were three main differences between these two techniques. First, BLRL calculated reuse distances only for accesses occurring within the cluster, whereas MRRL calculated reuse distances for all accesses within the skip region and cluster. Second, BLRL used reuse distances calculated solely in the skip region to update histograms, while MRRL updated histograms from the reuse distances determined from the skip region and cluster. Third, BLRL calculated warm-up from the reuse distances that crossed into the cluster boundary (i.e., they were accessed within the

cluster); MRRL calculated warm-up from all reuse distances. BLRL, therefore, estimated the reuse distances which affect cluster measurements directly to obtain warm-up lengths that directly conform to the desired level of accuracy and were based upon behaviors within the cluster.

The use of BLRL warm-up provided estimates with similar error rates as MRRL, with much shorter warm-up times. Tested reconstruction percentages of 85%, 90%, and 95% were used for BLRL; percentages of 99.5% and 99.9% were used for MRRL. For a number of SPEC2K workloads, 50 clusters were systematically placed every 100M instructions to estimate CPI. A cluster size of 1M instructions was used. The average error rates of BLRL-90% and MRRL-99.9% were 0.30% and 0.43%, respectively. Although BLRL achieved a lower absolute error rate, the differences were not statistically significant since both errors were well below the target margin of error. Warm-up lengths of BLRL-90% and MRRL-99.9% were 453M and 896M instructions, respectively. Although BLRL resulted in significantly lower warm-up lengths than MRRL, it was unclear if this reduction was due to the lower percentage of accuracy (i.e., 90% vs. 99.9%) used for the two techniques.

Van Ertvelde, et al. [77], proposed no state-loss boundary line reuse latency (NSL-BLRL), as an extension to BLRL. NSL-BLRL included checkpoints that stored *no-state-loss* [21] (NSL) cache reconstruction state. No-state-loss data consisted of a least-recently used unique cache accesses that occurred within the skip region prior to a cluster. By utilizing checkpoints that contained NSL state, the checkpoints were useful across many different cache configurations.

2.5.3 Multi-processor Warm-up

2.5.3.1 Memory Timestamp Record (MTR)

Barr, et al. [8] proposed a warm-up technique used to reconstruct cache and directory state for multiprocessor simulation. Utilizing a structure called the memory timestamp record (MTR), functional warming of cache accesses was replaced by faster MTR updates. An MTR entry was required for each physical block of memory to track coarse-grain access interleaving by individual processors. Each MTR entry contained three items: 1) a full bit-vector, *readers*, which tracked the last timestamped read access of every processor; 2) an ID of the last writer to the address, *writer*; and 3) the timestamp of the last writer to the address, *writertime*. Updates to the MTR structure are shown in Figure 6, and occurred within cluster execution[10] as well as during functional fast-forwarding. Cache and directory reconstruction was performed using data collected within the MTR structure at the beginning of each cluster. Since complete reconstruction of cache and directory structures is costly, the MTR was augmented with dirty bits to identify the modified MTR elements. Reconstruction could therefore be selectively applied to only the changed MTR elements. The simulated architecture was an SMP model. Processors had private L1 caches with LRU replacement. Memory was tracked using a full bit-vector based directory scheme using the MSI coherence protocol with write-back invalidation.

[10] Updates were performed for detailed simulation to ensure consistency between the MTR records and the actual state of physical memory. Otherwise, the MTR data structures would need to be updated to reflect changes during detailed execution.

```
update(time, address, isStore, cpu) {
    MTR[address].reader[cpu] = time;
    if (isStore) {
        MTR[address].writer = cpu;
        MTR[address].writetime = time;
    }
}
```

Figure 6: MTR Updates Caused by Processor Memory Requests

Cache reconstruction consisted of two phases. The first phase filled cache blocks of each processor's cache with the aid of the cache set record (CSR) data structure. Each CSR entry contained fields for the tag, timestamp, valid bit, and dirty bit. The CSR was used to organize MTR records into timestamp-sorted list representations of cache sets for all processor caches. For a *k*-way set-associative cache, the CSR contained the *k* most recently accessed references for each set. All accesses were inserted into the CSR irrespective if they were valid, since valid blocks later invalidated by the coherence protocol could cause evictions that must be modeled. After cache blocks were filled into the CSR, the second reconstruction phase involved modeling cross-processor interactions to determine the dirty and valid bits. Since writes invalidated any previously cached copies, accesses that occurred before the last writer were invalidated. Accesses that occurred equal to or after the last write time were valid. These references were resolved as clean or dirty. A block was clean if it was accessed (via a read) after the last writer since it would be downgraded into shared state; otherwise, the block was set dirty.

The directory reconstruction was performed using MTR data in a similar manner as cache reconstruction. If a cache block was never written, then the sharers list consisted of all processors that read the block. If a cache block was written, then sharers consisted of all processors that read the block after the last write time. Since the MSI

coherence protocol with write-back invalidation utilized a silent drop policy, evicted clean blocks did not require directory notification and remained in the sharers list. Conversely, evicted dirty blocks required directory notification. If the last access to a block was a write, the block was set to modified state and was the only sharer. The CSR structure was searched for dirty blocks that were valid (i.e., blocks that remained cached after possible invalidation from the second step of cache reconstruction). These blocks were set to modified state.

One major issue in applying sampling towards multithreaded architectural simulation involves the interleaving of threads during functional fast-forwarding. When functionally fast-forwarding between clusters, the interleaving of threads will differ from those in the full simulation. Unfortunately, even small perturbations in thread timings may result in large variations in simulation results [3]. In this study, the effects of divergent thread interleaving upon sampled results were compensated through random timing variations. Timing variability was achieved by selecting a different processor every 10,000 instructions to execute 25% slower than its peers. Such forced timing variations caused each sampled simulation to exhibit various representative thread interleavings and coherence races. Each simulation (full detailed and sampled) was performed eight times and the median values of performance metrics were used for accuracy comparisons.

The accuracy and runtime performance of MTR was compared with full functional warming. The simulation infrastructure utilized the Bochs [51] full-system emulator to drive a distributed shared memory model. The simulated architecture consisted of four processor CMP systems. Simulated workloads included a number of

multithreaded workloads from the Fortran/OpenMP NAS parallel benchmark suite. In multiprocessor simulations, IPC is not a representative metric of program performance. Therefore, sampling accuracy was evaluated through the comparisons of cache miss rates and coherence message counts (per 1,000 memory references). In all tested experiments, MTR closely approximated full functional warming. MTR also obtained speedup over full functional warming.

In the single-threaded domain, full functional warming is an extremely accurate warm-up method. In the multi-processor domain, however, full functional warming cannot guarantee similar accuracy. MTR, by extension, also has similar issues since its goal is to approximate full functional warming. At a sampling ratio of 1:100, sampled estimates for both MTR and full functional warming typically exhibited less than 15% error. At a sampling ratio of 1:1000, many workloads exhibited errors as large as 25%. In contrast, single-threaded sampled simulation can provide extremely accurate estimates even at sampling ratios of 0.1%, or lower. The high error rates observed in this study were due to divergent thread interleavings randomly introduced through core throttling. Although eight simulations were performed to determine the median performance values, the sampled simulations did not faithfully capture the thread interleavings observed by the full simulations. As core counts increase, the potential for thread interleavings to have greater impacts upon sampling accuracy also increases. It is currently unclear whether or not the proposed technique could be applied to manycore systems containing significantly more processors.

2.6 Thesis Contributions

2.6.1 Reverse State Reconstruction

The reverse state reconstruction (RSR) warm-up method is presented as an accurate and fast non-sampling bias removal technique [13]. While functionally skipping between clusters, the data necessary for reconstruction are recorded. After functional skipping has completed and the next cluster reached, processor state is approximated by functionally applying data in reverse order. By processing state in reverse order, non-sampling bias may be effectively reduced without functionally applying every record, as required by full functional warming. Although full functional warming is extremely accurate, many instructions are ineffectual and may be omitted without significant effects to non-sampling bias. Since reconstruction data are recorded, the identification of ineffectual instructions is performed without profiling information. The proposed scheme trades storage costs for simulation speed, and proposes on-demand state reconstruction for sampled simulations. RSR algorithms are developed and evaluated for cache and branch predictor warm-up. Compared to SMARTS [80], [81], [82], [84], Reverse State Reconstruction [13] achieves a maximum and average speedup ratio of 2.45 and 1.64, respectively. Simulation is a tradeoff between speed and accuracy. However, RSR obtains significant speedup improvements with negligible losses in accuracy (less than 0.3%).

2.6.2 Single-Pass Sampling Regimen Construction

The single-pass sampling regimen construction algorithm [12] is presented to quickly isolate and identify appropriate sampling regimen designs. Sampling regimen design refers to the selection of an appropriate cluster size and the number of clusters for

inclusion in a sample. Historically, sampling regimen design was an iterative process that required full workload simulations for error comparisons. The iterative process involved taking samples from the workload, evaluating them against the full simulation, and performing additional simulations until an appropriate sampling regimen was discovered. In many instances, the identification of a valid sampling regimen involved the arbitrary selection of sampling regimen parameters (e.g., trying different cluster sizes or random seeds, etc.). In contrast, the single-pass sampling regimen method allowed thousands of sampling regimen candidates to be simultaneously evaluated from a single simulation. With this technique, simulation speed was increased by an average factor of 17x with a maximum increase of 73x relative to the total workload simulation. Additionally, the single-pass sampling regimen technique allows the user to effectively estimate the true workload performance and the sample error without running the entire workload.

2.6.3 Enumerating Challenges that Currently Prevent Multi-threaded Sampling

The application of single-threaded sampling techniques to multi-threaded architectures is a non-trivial endeavor. In fact, all sampling strategies developed for single-threaded sampling cannot be directly applied to this class of computation. A discussion of the fundamental challenges that currently prevent the application of sampling to multi-threaded architectures is discussed. Identified issues include the identification of stable rate-based metrics for sampling to estimate, as well as the circular dependence dilemma. The circular dependence dilemma characterizes the situation where (while skipping instructions between clusters) thread progress is dependent upon system state, and vice versa. Future sampling strategies must overcome these issues.

Multi-threaded architectures also introduce a number of non-sampling bias components must be effectively reduced for reliable sampling methodologies (i.e., coherence state, directory state, NOC traffic and contention, unknown thread progressions, etc.). In order to measure the non-sampling bias component associated with divergent thread schedules, a novel quantitative thread skew metric was introduced. The implications associated with thread skew are discussed, including qualitative methods that may be used in its reduction.

2.6.4 Barrier-Interval Time-Parallel Simulation

A novel time-parallel barrier-interval simulation technique is presented to rapidly accelerate the simulation of certain classes of multi-threaded workloads. By segmenting a program into intervals delineated by barriers, simulations may be parallelized into time-discrete intervals and avoid many of the challenges currently preventing the accurate and reliable sampling of single-application, multi-threaded workloads. Most notably, if simulation begins immediately following a barrier release, then the proper thread interleavings are approximately known. The simulator modifications necessary to support barrier-interval simulation are minimal, and are likely implemented in many architectural simulators. For the workloads tested, wall-clock speedups ranged between 1.2x to 596x, with an average speedup of 14x. Furthermore, barrier-interval simulation allows for the measurement of stable performance metrics such as cycle counts with minimal losses in accuracy (2%, on average). Barrier interval simulation provides architects with a fast and accurate mechanism to rapidly accelerate particular classes of manycore simulations.

CHAPTER 3

REVERSE STATE RECONSTRUCTION FOR SAMPLED MICROARCHITECTURAL SIMULATION

3.1 Reverse State Reconstruction

While skipping between clusters, many instructions do not affect the final processor state immediately prior to the next cluster. If such instructions can be identified, they can be safely omitted during warm execution with no adverse effects on sampling accuracy. For example, cache blocks generated by memory operations at the beginning of a skip region will likely be evicted or modified during gap processing. The *Reverse State Reconstruction* algorithm reconstructs the state of the cache and branch predictors by iterating through a logged skip-region trace in ***reverse order*** and judiciously applying updates as needed. After a cluster is finished, cold-phase execution advances simulation to the next cluster boundary. During this phase, certain instructions are logged from the functional simulator. Branch information is saved for reconstruction of the branch predictor, and memory operations are saved for the caches. To minimize the storage requirements of the algorithm, data are kept only for the current cluster of execution. When the current cluster finishes, any saved information is discarded to accommodate data in the next skip region. The contributions of this technique are threefold:

1. The proposed method isolates ineffectual instructions from skip regions between clusters without the use of profiling.
2. The proposed method achieves a maximum and average speedup ratio of 2.45 and 1.64, respectively, over SMARTS with minimal sacrifice to accuracy (less than 0.3%).
3. By trading storage for speed, the proposed method introduces the concept of on-demand state reconstruction for sampled simulations**.**

The two most important elements to warm-up in sampled processor simulation are the branch predictor and cache hierarchy. The following sections describe how these structures are reconstructed.

3.2 Reverse Cache Reconstruction

Cache reconstruction begins by logging memory operations during cold simulation. During logging, the state of the cache is left *stale*. A stale cache contains the same data present after the execution of the previous cluster. The current PC, next PC, the address of the data or instruction, and two Boolean values specifying the entry type (instruction or data) and reference type (load or store) are buffered. Immediately before the next cluster, the reference stream is scanned in reverse order and the cache state is updated. Temporal locality is exploited by applying updates to the cache for only those references which would have affected the final state. References that occurred at the beginning of the skip region, which subsequently are evicted by future references, can be safely removed from warm-up. Redundant references (i.e., references to data already reconstructed in the cache) can also be ignored since their effect on the set has been previously processed. Each cache block contains a bit indicating if it has been

reconstructed. These bits are cleared before the logged data are used to warm cache state. Whenever a cache block is reconstructed, its associated reconstructed bit is set.

For each logged reference, a lookup is performed to determine if its corresponding set has been reconstructed. *Since the logged data are processed in reverse order, redundant accesses to a reconstructed set actually occurred earlier and can be ignored*. If the set has been reconstructed, then all subsequent accesses can be ignored. If not, then the reference is classified as present or absent. If present, the LRU bits are updated only if they are stale. If absent, the reference is inserted into the least recently used stale block. After deciding where the block should be placed, the LRU bits of the reconstructed blocks are assigned in ascending numerical order. The first reconstructed block of a set is assigned the most recently used reference. Additional unique references reconstructed into the same set are assigned increasing LRU values. The last reconstructed block of a set is assigned the least recently used reference.

As the logged trace is consumed, sets within the cache are reconstructed. References to reconstructed blocks can be safely ignored since they do not affect the final cache state. Experiments demonstrate reverse state reconstruction can sufficiently approximate the cache state without functionally simulating the cache for every reference in the skip region. Despite the buffering of the data reference stream during functional simulation, numerous cache updates are avoided resulting in faster simulation times.

Figure 7 shows an example of the reverse cache reconstruction algorithm. In this figure, the number above the cache block indicates its LRU value. The letter R indicates that the block has been reconstructed. In this example, a forward reference stream E, A, F, C is applied to a particular cache set. Two columns are used to show the

reverse state algorithm approximates regular cache simulation for a particular cache set. The left column shows normal cache simulation. As the references are applied on the left, the least recently used element is evicted from the cache, and replaced by the incoming reference. The newly placed reference is the most recently used element, and the LRU bits of the other blocks in the set are updated. After all accesses have been applied, the final state of the cache set is shown.

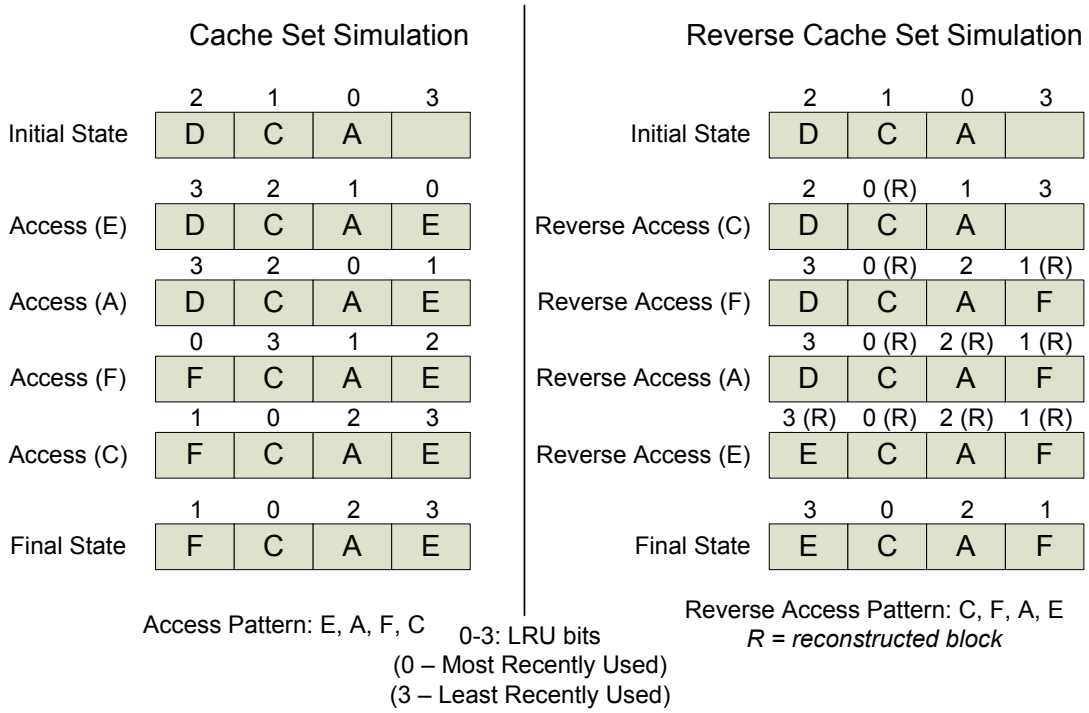


Figure 7: Reverse Cache Reconstruction of an Individual Cache Line

On the right column of Figure 7, the details of the reverse cache reconstruction method are shown. The LRU bits for stale elements are used to determine an appropriate way for block insertion. Reconstructed blocks are placed into the *least recently used*

stale element. The set is searched for the maximum LRU value for all reconstructed blocks. If no blocks have been reconstructed, the newly reconstructed block becomes the most recently used. If blocks have been reconstructed, the LRU values will increase. The last reconstructed block becomes the least recently used. As shown in Figure 7, *Reverse Cache Reconstruction* closely approximates normal cache simulation.

3.3 Reverse Branch Predictor Reconstruction

Branch Predictor reconstruction involves state repair in the prediction tables, branch target buffer (BTB), and return address stack (RAS). Branch predictor reconstruction begins by logging branch information during cold simulation. Buffered data include the current PC, next PC, branch outcome, and other accounting information relevant to determine the final branch effects (such as the instruction opcode, source register, and instruction flags). A BTB element in the branch predictor is reconstructed using the address logged during functional simulation.

Unlike cache reconstruction, the branch predictor is updated *on-demand* within the next cluster of execution. Specifically, as branches are encountered in the next cluster, the branch predictor is probed to determine if the entry has been reconstructed. If the entry has been reconstructed, then execution in the cluster continues normally. If not, the entry is first reconstructed before hot execution continues. During the traversal, branches that reference entries not relevant to the current entry (i.e., branches that do not index into the same table entry) are also reconstructed. By reconstructing other branches in this manner, the logged data need not be rescanned from the beginning for each uniquely indexed branch. Since a Gshare predictor is used, the global history register must first be reconstructed using the last n branches of the skip-region trace (where n is

the width of the global history register). Once the global history register has been reconstructed, branch entries can be accurately determined. Like cache reconstruction, the contents of the branch predictor are left stale prior to reconstruction.

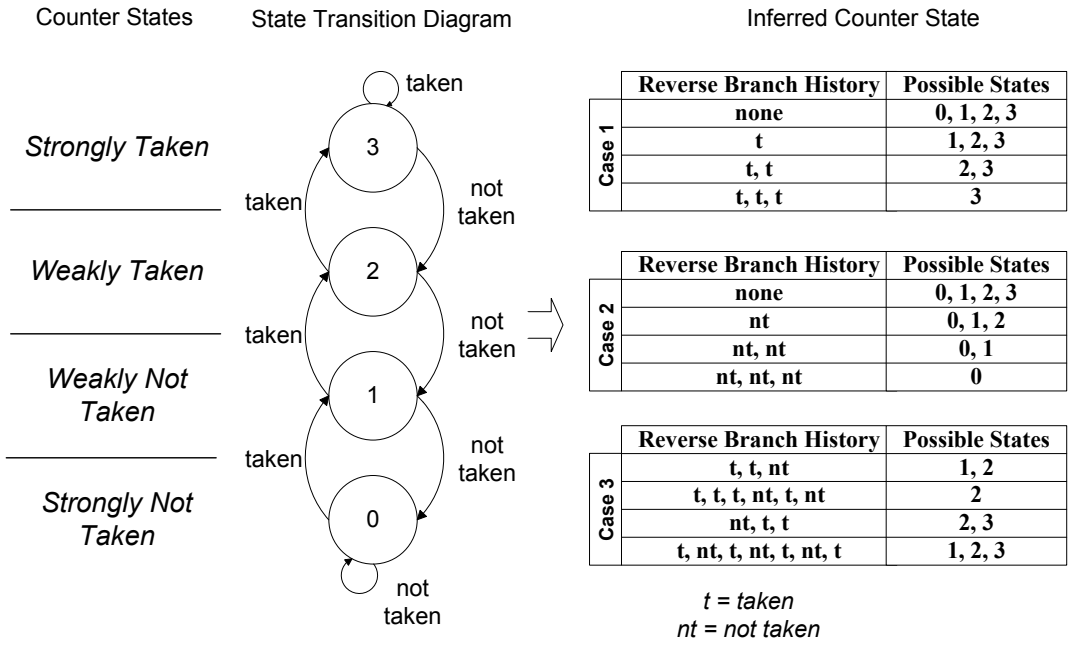


Figure 8: Reverse State Reconstruction of Branch Table Entries

Figure 8 shows the normal operation of a 2-bit saturating counter entry indexed within a branch predictor. Each counter value indicates a prediction state. When an instruction is retired, the initial prediction is updated with its outcome. Taken branches cause the counter to increment, and not taken branches cause the counter to decrement. Since the 2-bit counter has a limited number of values, usually only a small amount of history is needed to approximately reconstruct a particular branch predictor entry. In other words, the logged branch history can be used to sufficiently isolate the exact

counter value, or narrow the counter value to a set of possible states. Examples of branch predictors with 2-bit counters include Yeh's algorithm, gshare, hybrid and multihybrid, skewed, and agree predictors [78].

During reconstruction, a series of possible states are tracked for each prediction table entry. Initially, the set of possible states includes all possible counter values: 0, 1, 2, or 3. As references to the same entry are encountered, a reverse branch history is generated. *The reverse branch history field in the table constitutes branch history for a particular set in the reverse order*. Therefore, the first outcome in the reverse history is the last outcome for that branch table entry in the skip region. The logged branch history is searched until the counter state for the branch can be determined or until the history has been consumed. Rather than performing this computation at execution time, a table was built *a priori* so that reconstruction could be implemented through a table lookup.

Figure 8 shows several examples of how the reverse branch histories isolated for a particular entry can be used to infer a branch counter or set of branch counters. If the last three consecutive outcomes for a particular branch entry are taken (or not taken), the exact counter state can be determined. Regardless of the initial counter state, three consecutive taken branches will cause the counter state to become three (strongly taken), and three consecutive not taken branches will cause the counter state to become zero (strongly not taken; see cases 1 and 2 in Figure 8). Furthermore, if these patterns exist anywhere within the branch history, then the exact counter state can also be determined (see case 3 in Figure 8). However, the branch history does not always yield an exact counter state. Case 3 shows instances where the exact state cannot be inferred. In this instance, the outcome is predicted based on the remaining set of possible states. If the

branch is biased in one direction (taken or not taken) the predictor is set to the weak form. If three states exist, the middle state is predicted. For example, if the remaining possible states include strongly not taken, weakly not taken, and weakly taken, then the state of weakly not taken is predicted. No more than three states can exist for an entry that has a history of one branch. If no history for a branch is produced, then the counter value is left stale.

Reconstruction of a finite size return address stack is accomplished through the following algorithm. Whenever a pop is encountered in the reverse history, a single counter is incremented. If a push is encountered, and the counter is equal to zero, the next PC is placed at the end of the RAS; otherwise, whenever a push is observed, the counter is decremented. Once the return address stack has been filled, reconstruction is complete. Figure 9 shows an example of a forward and a reverse call sequence. The numbers next to the reverse call sequence indicate the counter value after the push/pop has been processed.

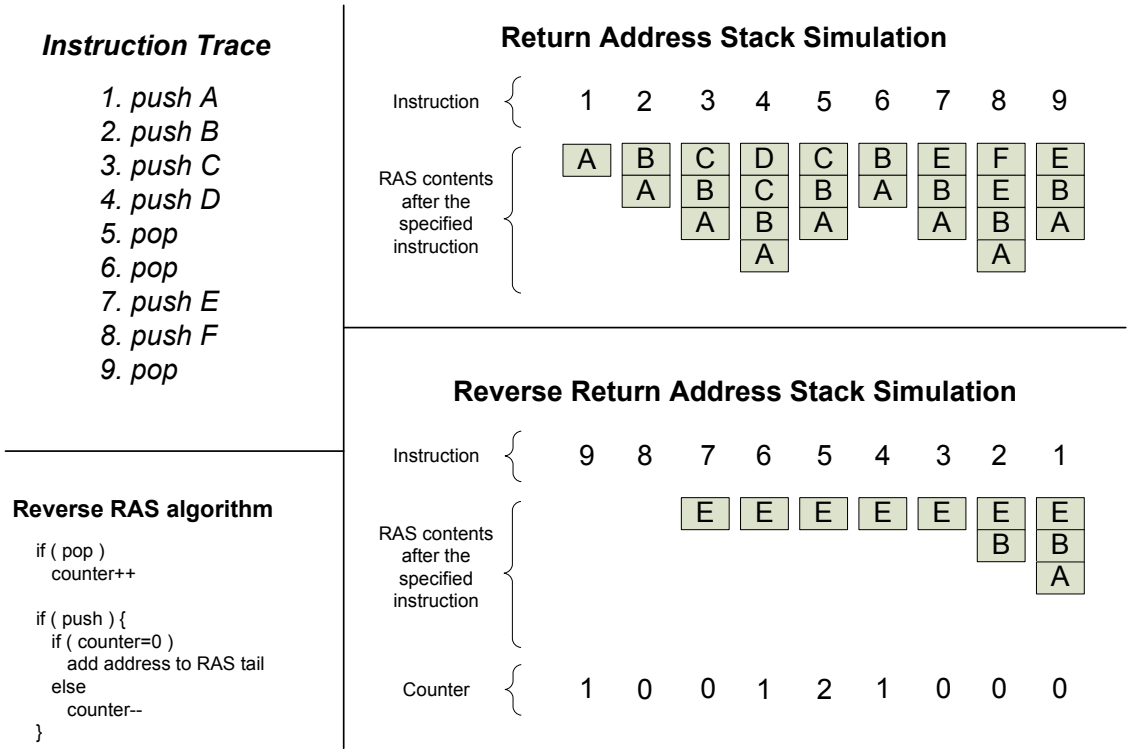


Figure 9: Reverse State Reconstruction for the Return Address Stack

3.4 Experimental Framework

The model used in this study is an execution-driven simulator based on SimpleScalar [14]. Unlike trace-driven simulation, the processor model fetches instructions from a compiled binary. The front end of the processor can fetch and dispatch eight instructions per cycle, and can issue and retire four instructions per cycle. The model includes eight fully pipelined universal function units. The maximum number of in flight instructions is 64. The issue queue size is 32, and the load-store queue is 64 elements. The pipeline depth is seven stages. The minimum branch miss-prediction penalty is five cycles. The processor frequency is assumed to be 2 GHz. The branch predictor is a 64K entry Gshare with an eight-entry return address stack. The BTB

contains 4K entries. Architectural checkpoints are utilized to allow the processor to speculatively execute beyond eight branches.

A substantive memory hierarchy is modeled within the simulator. The first level data cache is 4-way and contains 32 KB with a 64-byte line size. The first level instruction cache is also 4-way and contains 64 KB with a 64-byte line size. The instruction and data caches are implemented using a write-through no-write allocate policy. The second level cache is 8-way and contains 1 MB with a 64-byte line size, and is implemented using a write-back write-allocate policy. A bus model is also incorporated in order to emulate arbitration, contention, and transfer delay between the levels of memory. The first level bus is shared between the first level data and instruction caches, and connects the first level caches to the second level cache. The first level bus has a width of 16 bytes and operates at 1GHz. The second level bus connects the second level cache to main memory, has a width of 32 bytes, and operates at 2 GHz.

The model includes both a functional and a timing simulator. The functional simulator is useful for a variety of reasons. First, the functional simulator is used to validate the results of the timing simulator. If the timing simulator attempts to commit a wrong value, the functional simulator will assert an error. However, in the context of sampled simulation, the functional simulator has additional uses. Second, as instructions in the dynamic stream are skipped (either in cold or warm simulation), the functional simulator retains valid architectural state. When hot execution continues in the next cluster, the values of the registers contained in the functional simulator are copied to the timing simulator.

For processor simulations, the standard performance metric is IPC, which is measured as the number of instructions retired per execution cycle. The *Reverse State Reconstruction* algorithm described above was tested against a number of other techniques for accuracy, speed, and statistical confidence.

3.5 Experimental Results

Experiments were conducted using the SPEC2000 benchmarks. Integer benchmarks used include *gcc*, *mcf*, *parser*, *perl*, *twolf, vortex*, and *vpr*. Floating point benchmarks used include *ammp* and *art*. The first six billion instructions from each benchmark were simulated with reference input sets. Table 4 shows the true IPC of each benchmark simulated during experimentation. The true IPC served as a baseline for comparison to the various sampling techniques. Sampling regimens were constructed for each workload and are included in the table. All sampling techniques from each compared benchmark utilized the specified sampling regimen, also shown in Table 4. The starting positions of each cluster were then randomly generated according to a uniform distribution. Identical starting cluster positions were used for each sampling algorithm (except SimPoint) to ensure constant sampling bias.

Table 4: True IPC and Sampling Regimen Data for each Workload

Benchmark	True IPC	Number of clusters	Cluster size
ammp	0.24811	1,024	1,000
art	0.77980	512	1,000
gcc	0.87314	1,000	10,000
mcf	0.20854	1,000	9,000
parser	1.07389	2,048	5,000
perl	1.28956	600	21,000
twolf	0.97398	1,024	1,000
vortex	0.92672	1,000	2,000
vpr	1.18062	256	6,000

Using this framework, a number of different techniques were compared to measure the effectiveness of non-sampling bias removal. As discussed previously, non-sampling bias is caused by the loss of state information during skipped periods. After a cluster is executed and instructions are skipped, the potential for state loss is high and will likely affect the performance of the subsequent cluster. Processor state is contained in a number of areas including: the scheduling queues, the reorder buffer, the functional unit pipelines, the branch prediction hardware, instruction caches, data caches, load/store queues, and control transfer instruction queues.

Each warm-up method or policy was then passed through a 95% confidence interval test in order to determine if it correctly predicted the true IPC. The standard deviation, S_{IPC}, and standard error, $S_{\overline{IPC}}$, for a cluster sampling design is given by,

$$S_{IPC} = \sqrt{\frac{\sum_{i=1}^{N_{cluster}} \left(\mu_{IPC}^{i} - \mu_{IPC}^{sample}\right)^2}{N_{cluster} - 1}}, \qquad S_{\overline{IPC}} = \frac{S_{IPC}}{\sqrt{N_{cluster}}}$$

where μ_{IPC}^{i} is the mean IPC for the i_{th} cluster in the sample. The estimated standard error was used to calculate the *error bounds* and *confidence interval*. Using the properties of the normal distribution, the 95% confidence interval is given by $\mu_{IPC}^{sample} \pm 1.96 S_{\overline{IPC}}$, where the error bound is $\pm 1.96 S_{\overline{IPC}}$. A confidence interval of 95% implies that, under repeated sampling, the true population mean will be contained within the calculated intervals with a 95% probability. Low standard errors imply relatively small variation in repeated estimates and consequently result in higher precision. For each warm-up policy, the relative error was calculated as follows:

$$RE(IPC) = \frac{\left|\mu_{IPC}^{true} - \mu_{IPC}^{sample}\right|}{\mu_{IPC}^{true}},$$

where μ_{IPC}^{true} was the IPC of the true population mean, and μ_{IPC}^{sample} was the estimate of the IPC obtained from the sample. The calculation of relative error relies upon the IPC from a full (un-sampled) simulation of each tested benchmark.

Table 5 shows the various warm-up methods used during experimentation. In *no warm-up*, no state repair techniques were used in the processing of skip region instructions. After the execution of a cluster, the caches and branch predictor were left stale. In *fixed period warm-up*, a specified percentage of the skip regions immediately prior to the next cluster were used for warm-up. Three variations of SMARTS warm-up were also conducted. The first two consisted of selectively warming only the cache hierarchy or branch predictor. These simulations were used to determine the accuracy of the *Reverse State Reconstruction* algorithms when selectively applied to the cache and branch predictor in isolation. The third variant of SMARTS warmed both the cache and branch predictor for comparison when the reverse state algorithm also warmed the cache and branch predictor together. All warm-up methods requiring specified percentages were performed using 20%, 40%, and 80%. Finally, a detailed comparison with SimPoint also was performed.

Table 5: Warm-up Method Experiments

Experiment		Description
None	None	No warm-up was performed during cold execution.
Fixed Period	FP	A specified percentage of the skip region was used for warm execution.
SMARTS cache	$S_{\$}$	All memory operations were functionally applied to the cache in the skip region.
SMARTS branch	S_{BP}	All branches were functionally applied to the cache in the skip region.
SMARTS cache/branch	$S_{\$/BP}$	All branches and memory operations were functionally applied in the skip region.
REWIND cache	$R_{\$}$	Reverse State Cache Reconstruction for a specified percentage of the skip region.
REWIND branch	R_{BP}	Reverse State Branch Reconstruction for a specified percentage of the skip region.
REWIND cache/branch	$R_{\$/BP}$	The combined Reverse State Reconstruction algorithm.
SimPoint	SimPoint	Multiple simulation point analysis utilizing SimPoint v3.2.

Each of the tested warm-up methods were compared based on accuracy, speed, and statistical confidence. Since the data were too voluminous to compare each individual benchmark, the average performance for each technique was shown. Specific workloads are discussed in greater detail. For the interested reader, all data used to create the graphs are included in the appendix at the end of this chapter.

Figure 10 shows the relative error and simulation time results for all simulations that selectively warmed only the caches. As shown, the *Reverse State Cache Reconstruction* algorithm closely approximated SMARTS cache warm-up. The average relative error for SMARTS cache was 3.1%, while the reverse cache warm-up was approximately 3.3%. Although error rates obtained from cache warm-up are highly similar, the simulation times varied significantly. Full functional simulation of the cache state required an average of 1443 seconds, while the 20% reverse cache warm-up required 1086 seconds. By applying the last 20% of the memory references to the cache hierarchy a speedup ratio of 1.41 was achieved for cache warm-up. For these simulations, *gcc* had the largest speedup ratio of 1.93 while *parser* had the smallest speedup ratio of 1.03. Therefore, reverse cache reconstruction always reduced simulation time when compared to SMARTS. As the warm-up percentages increased, the speedup ratio was degraded. At 40% and 80%, the speedup ratios were 1.27 and 1.05, respectively. At 40%, most workloads performances were improved, however *mcf* exhibited degradation in simulation speed with a speedup ratio of 0.97. At 80%, all workloads showed speedup except *mcf*, *parser*, and *vortex*. Minimal benefit was obtained by executing more than 20% of the logged cache data. This is consistent with

temporal locality, such that the cache blocks at the beginning of the skip-region are likely to be evicted by subsequent references.

Figure 11 shows the relative error and simulation time results for all simulations that selectively warm-up only the branch predictor. As shown, the *Reverse State Branch Predictor Reconstruction* algorithm performed similarly to SMARTS. Both the reverse algorithm and SMARTS warm-up achieved an average relative error of 22.3% and 22.2%, respectively. However, the average speedup ratio of the reverse technique over SMARTS branch prediction warm-up was 1.48. *Gcc* exhibited the highest speedup ratio of 2.26, while *mcf* had the lowest of 1.10.

As shown in Figures 5 and 6, the cache hierarchy component contributed the greatest non-sampling bias for sampled simulation. Warming the branch predictor in isolation produced an average relative error of 23% while warming the cache in isolation produced an average relative error of 3.1%. Although it may appear advantageous to only warm the cache structures in sampled simulation, non-sampling bias produced by cold state in the branch predictor is sufficient to cause many simulations to fail confidence interval tests (see appendix).

Figure 12 shows the relative error and simulation time results for all simulations that incorporated both the cache and branch predictor in warm-up. No warm-up had the least overhead of all techniques, and thus had the lowest simulation time but produced the highest error at 23%. Of the remaining techniques, SMARTS had the lowest error at 0.9%, but had the highest simulation time. *Reverse State Reconstruction* achieved speedup ratios of 1.64, 1.51, and 1.25 for 20%, 40%, and 80%, respectively. At 20% and 40%, all workloads simulated faster with reverse state reconstruction than with

SMARTS. At 80%, *mcf* was the only workload which exhibited a slowdown in simulation time with a speedup ratio of 0.92

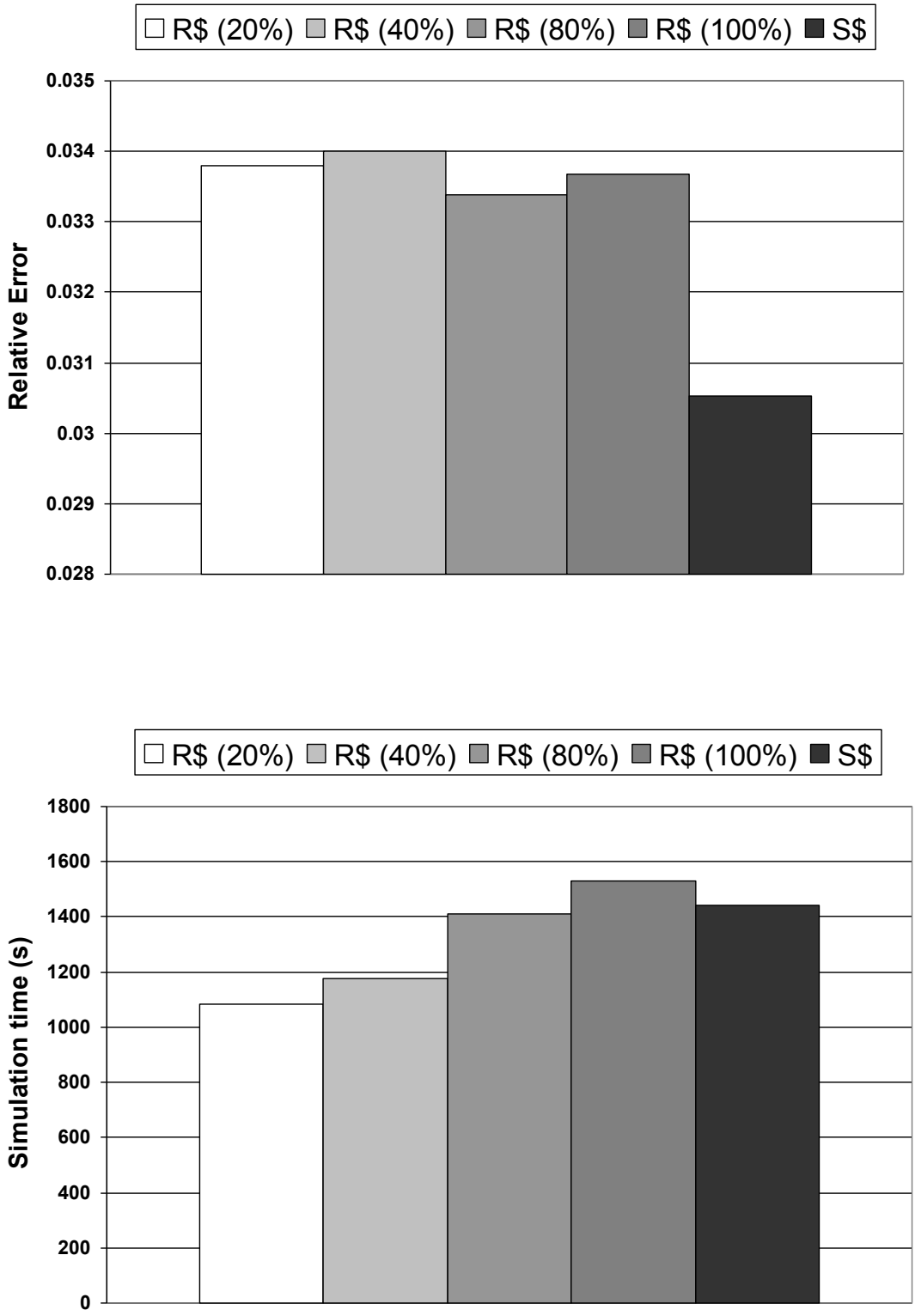


Figure 10: RSR Cache Warm-up Results

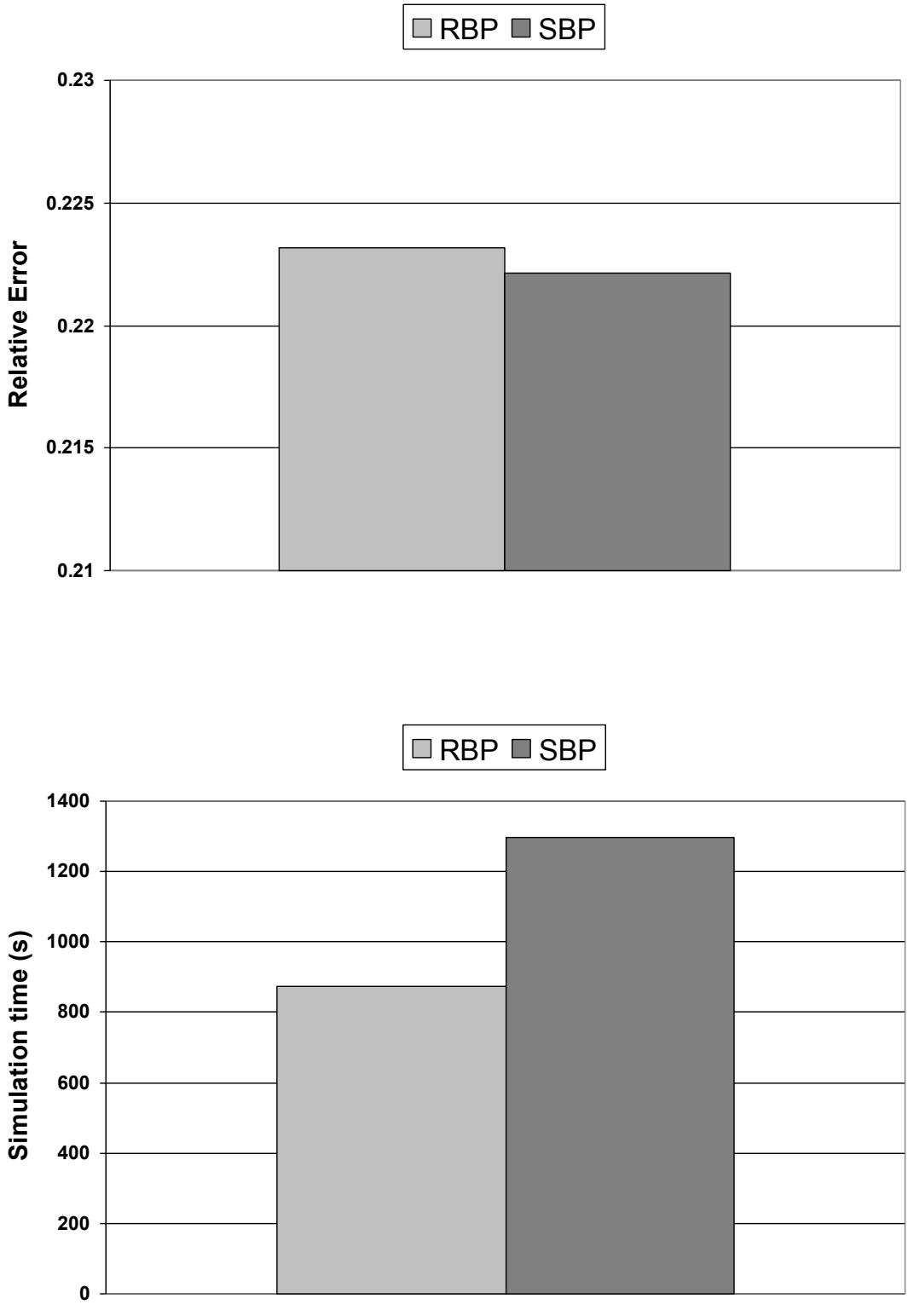
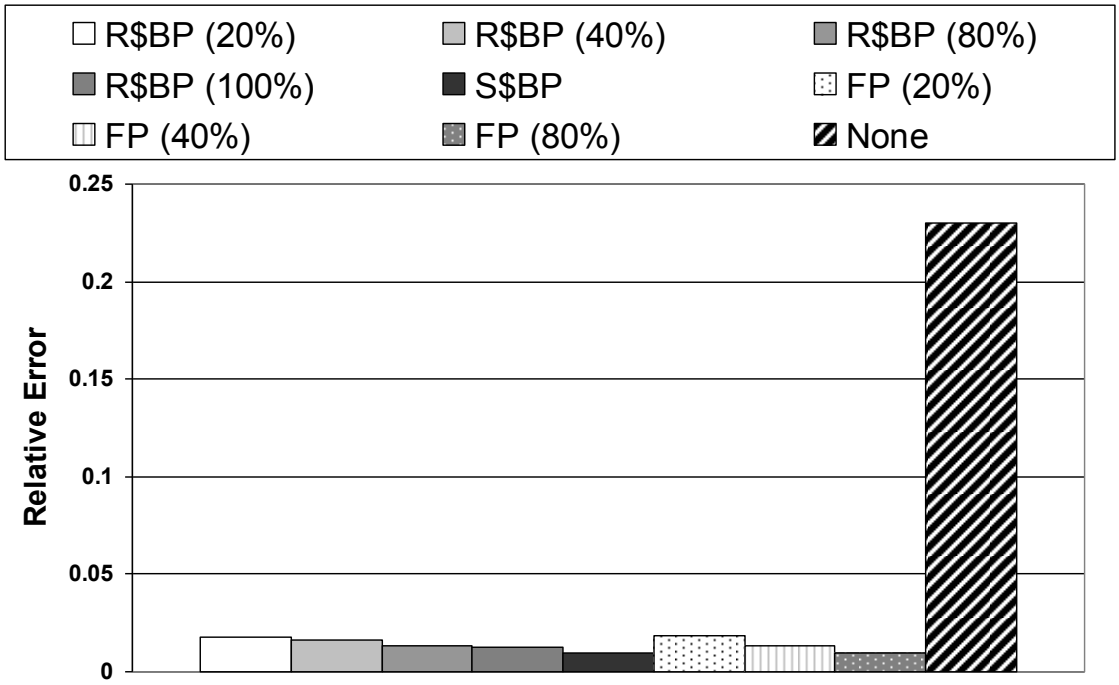


Figure 11: RSR Branch Predictor Warm-up Results



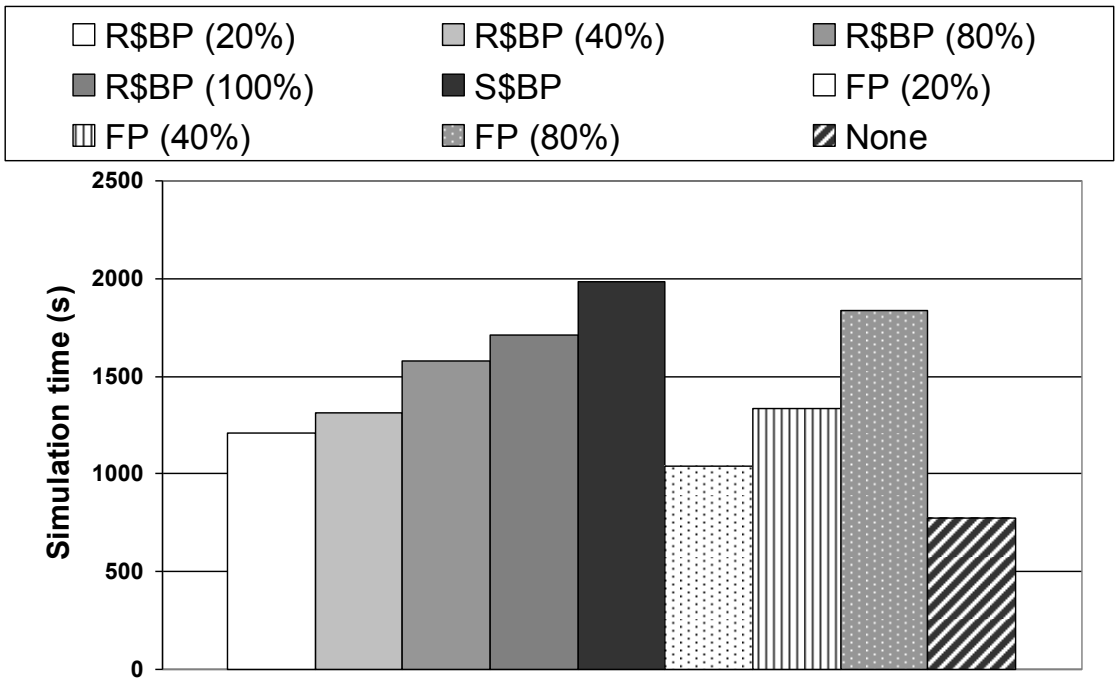


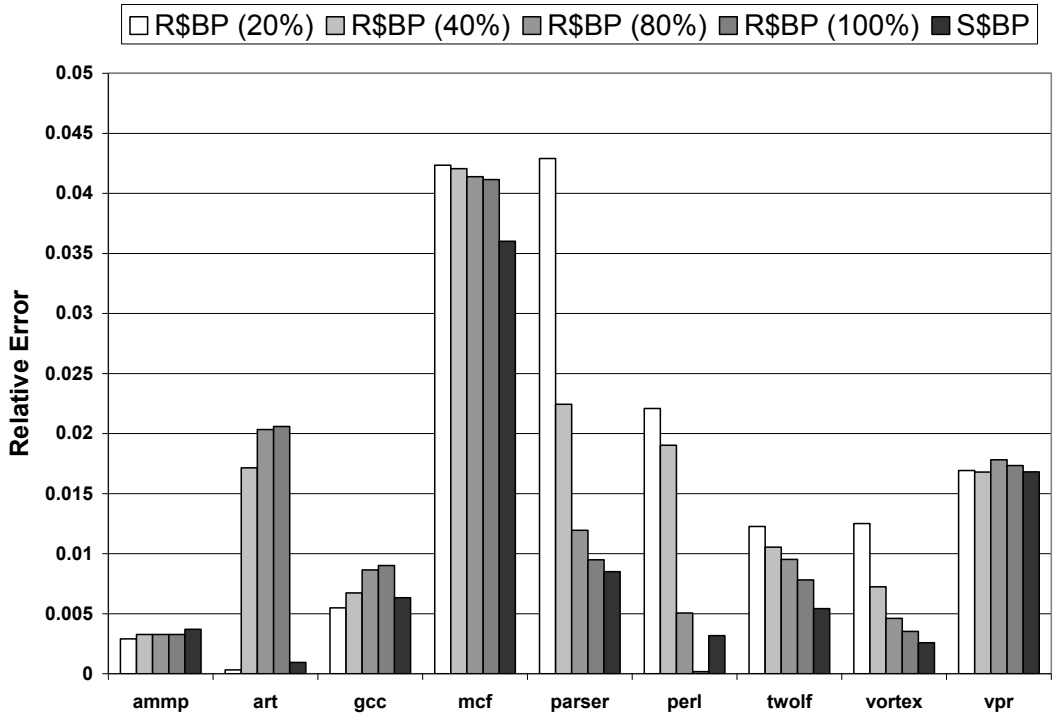
Figure 12: RSR Cache and Branch Prediction Warm-up Results

Fixed period simulations performed similarly to reverse warm-up at the specified percentages. At 20%, fixed period had a lower simulation time. However, as the percentages increased to 40% and 80% the reverse techniques were faster. One explanation is that all accounting information necessary for reconstruction was logged in the skip region, regardless of the warm-up percentage. As the reconstruction percentages increased, data buffering costs were amortized over the reconstruction time.

Figure 13 shows the relative error and simulation time results for the *Reverse State Reconstruction* compared to SMARTS warm-up. At 20% warm-up, the average relative error with respect to SMARTS for all simulated workloads was 0.3%. At 20% warm-up, minimum and maximum relative errors with respect to SMARTS were 0.01% and 1.90%, respectively. Since SMARTS is one of the most accurate warm-up methods, it was expected SMARTS should have the lowest error. Figure 12 shows the average behavior for the tested workloads. Figure 13 shows these results by individual benchmark. As expected, the simulation time increased as the specified warm-up percentage increased. Figure 14 shows the average relative error and simulation times for SimPoint with the *Reverse State State Reconstruction* at 20%. In order to fairly compare SimPoint with sampled simulation, a variety of different interval sizes were incorporated. All SimPoint comparisons were conducted utilizing multiple simulation points (30), at varying interval sizes. SimPoint v3.2 [73] was used in these experiments.

SimPoint allows the user to specify an interval size defining the basic block vector profiling granularity. Originally, an interval size of 50K was selected to keep the number of instructions in hot execution constant. As shown below, SimPoint produced

an average error of 20% when an interval size of 50K was used. One reason for the higher error rates is the SimPoint algorithm did not incorporate warm-up while skipping to the next simulation point, or cluster. Without warm-up, measurements taken from small clusters were greatly affected by non-sampling bias. Therefore, SMARTS warm-up was incorporated into the SimPoint simulations. In 50K-SMARTS, the SMARTS warm-up policy was used to warm-up processor state while skipping instructions to the next simulation point indicated by SimPoint. As shown below, the error rate dropped to 8% when warm-up was included.



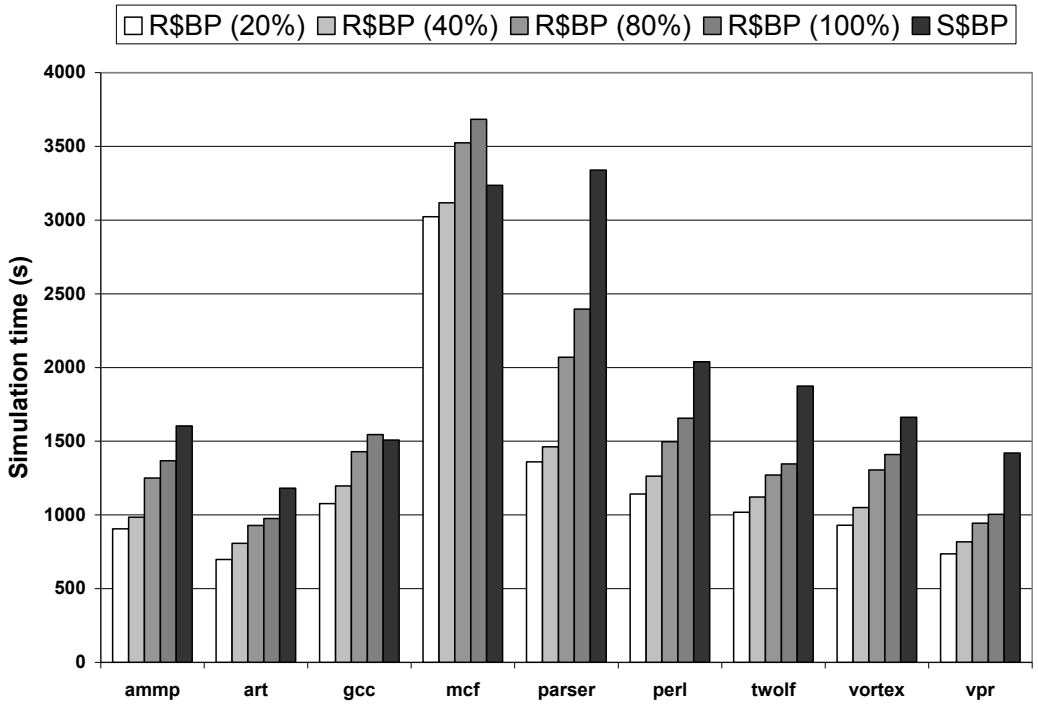
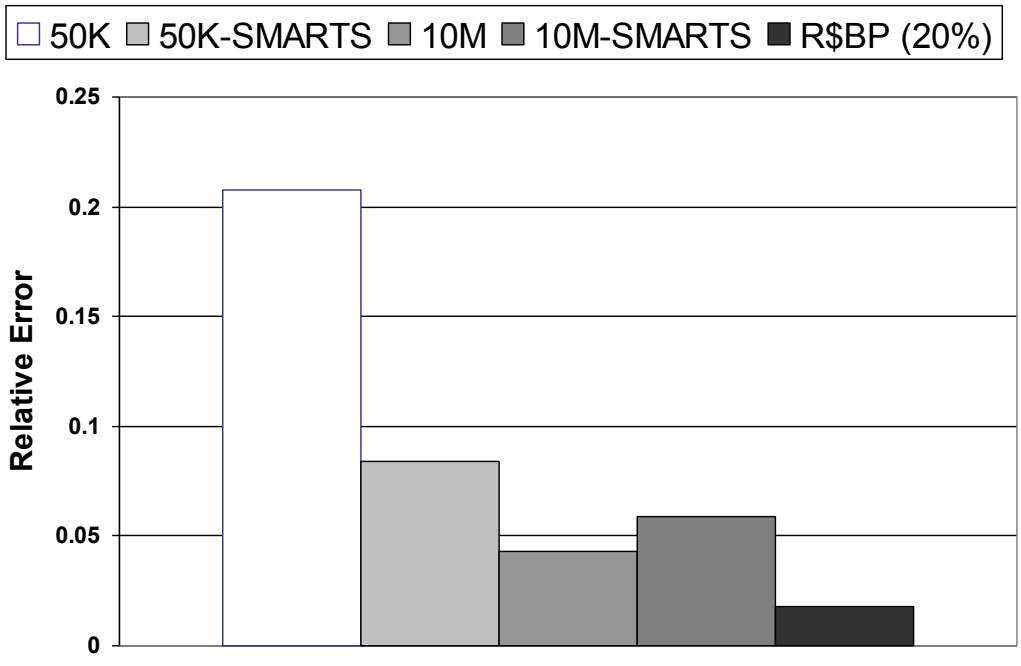


Figure 13: RSR vs. SMARTS

The authors of SimPoint, however, do not advocate small simulation point sizes (i.e., 50K instructions) be used. Therefore, the interval size was increased to a much larger size of 10M instructions. Using a 10M-interval size, the relative error of SimPoint decreased significantly to 4.2%. For a symmetrical comparison, SMARTS warm-up with an interval size of 10M was also performed, and had an average error of 5.9%. With an interval size of 50K the introduction of a warm-up method helped simulation accuracy. However, with an interval size of 10M its accuracy was degraded. No conclusions could be drawn from the addition of warm-up to the SimPoint method.

At the lowest interval size, SimPoint was faster than sampled simulation; however, it also exhibited higher error. Increasing the interval size increased accuracy, but resulted in longer running simulations. In contrast, the *Reverse State Reconstruction* algorithm had the lowest error (1.7%, on average) and resulted in faster simulation times than all but SimPoint-50K (which exhibited the highest error).

All warm-up methods were tested for statistical confidence (see appendix at the end of this chapter). Using a 95% confidence interval, the variability of each sample was tested to determine if it correctly predicted the true IPC. At 20% warm-up, the reverse state reconstruction correctly predicted the true IPC for seven of the nine workloads. The remaining two workloads were also predicted at 80% warm-up.



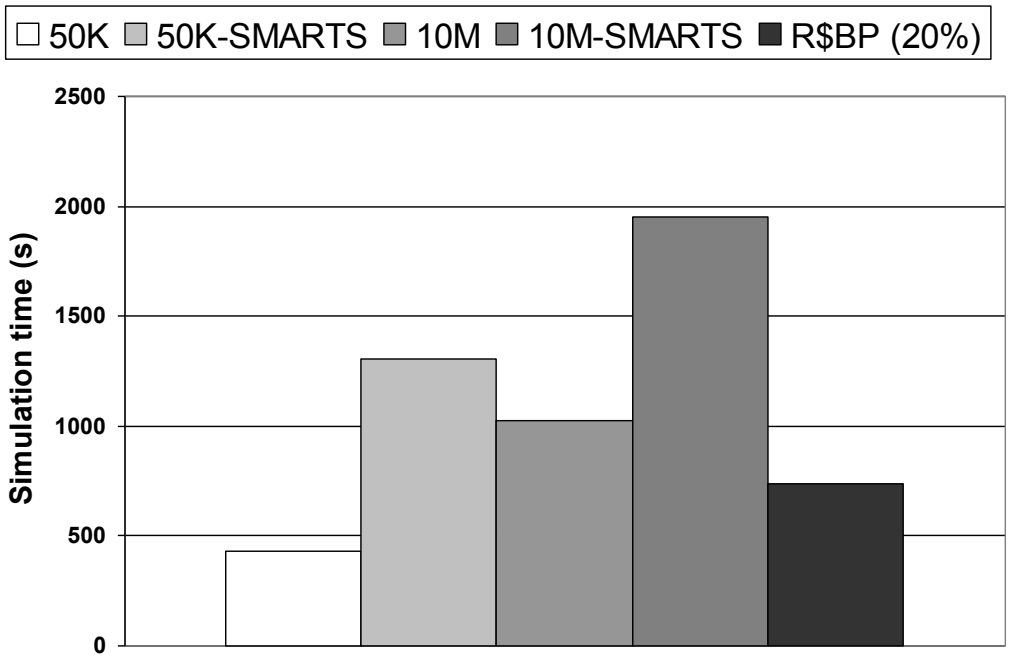


Figure 14: RSR vs. SimPoint

3.6 Related Work

Many different approaches have been used to remove non-sampling bias from sampled simulation. Laha, et al. [49], took sampling units immediately following context switches to ensure consistent state. By assuming the cache contents were flushed after a context switch for small caches, the contents were empty and therefore identical to the full execution trace. For larger cache designs, primed cache sets were introduced by Fu, et al. [30] and Laha, et al. [49]. After the execution of a new cluster began, a set in the cache was considered primed after it was filled with unique references. Only information gathered from primed sets were used to record measurements. The *Reverse State Reconstruction* [13] algorithm for cache warm-up is similar to primed set reconstruction. Before a cache set was simulated in the next cluster, its state must be reconstructed. Conte, et al. [23] extended the idea of primed cache sets to processor simulation by using stale state with a specified percentage of cluster instructions for cluster warm-up. Other warm-up techniques proposed by Wood, et al. [83] used probability functions to classify misses at the beginning of a cluster (unknown references) as either *compulsory* or *cold-start* misses.

Of all of the warm-up methods, perhaps the most accurate in removing non-sampling bias is SMARTS [84], proposed by Wunderlich, et al. When skipping instructions between clusters, the entire skip region of instructions is executed in a warm phase. Thus, every branch and memory operation is functionally applied to the branch predictor and cache hierarchy. The SMARTS warm-up policy has been applied in cache simulations [21], [22], [30], [49], [80], [81], [82], [84] and to processor simulations [23], [50], [71]. The SMARTS, or full functional, warm-up method is extremely accurate, but

at a cost. SMARTS incurs higher costs since many instructions within the skip region do not affect the final state prior to the next cluster. The *Reverse State Reconstruction* algorithm is able to dynamically isolate and remove such ineffectual instructions to realize faster simulation times than SMARTS.

Since SMARTS is demanding in terms of simulation time, other warm-up methods have been proposed to approximate SMARTS accuracy at a lower cost. Haskins, et al. [34] proposed the *Memory Reference Reuse Latency (MRRL)* algorithm for warm-up. MRRL profiles the skip regions between clusters to determine the number of pre-cluster instructions to execute for a given percentage of accuracy. This work was later extended by Eeckhout, et al. [29] with the *Boundary Line Reuse Latency (BLRL)* algorithm. Unlike MRRL, BLRL considers memory references from instructions that originate within the cluster. In this study, cluster and pre-cluster pairs are profiled for memory references. Only references in the pre-cluster that affect memory operations in the cluster are applied to the cache. The *Reverse State Reconstruction* algorithm proposed in this paper, unlike the aforementioned techniques, requires no profiling or analysis of skip region instructions. Although effective, the MRRL and BLRL techniques are dependent upon specific cluster locations and require additional profiling whenever the cluster positions are changed. Checkpoints have also been introduced [80], [82] using SMARTS warm-up in order to reduce the cost of functional skipping. However, checkpoints are also dependent upon specific cluster locations. NSL-BLRL [77] is an extension to BLRL that only checkpoints referenced data within each cluster. Thus, changing the cluster size would also require new checkpoint generation.

Vengroff, et al. [78] proposed a similar mechanism to warm prediction table counters by backtracking the DFA of a 2-bit saturating counter. In this work, branch histories were used to perform a single-pass trace simulation of multiple branch predictor configurations. This work extends [78] by adding reverse warm-up for the BTB and RAS components in order to remove non-sampling bias from the branch predictor in processor simulation environments.

3.7 Conclusion

In this chapter, a new *Reverse State Reconstruction* warm-up method was introduced for sampled simulation. Using this method, considerable speedups were achieved relative to SMARTS, with negligible accuracy losses. Maximum and average speedup ratios of 2.45 and 1.64, respectively, were obtained with accuracy losses of less than 0.3%, on average. By recording data while skipping instructions, processor state can be reconstructed *on-demand* rather than naively applying every memory operation and branch instruction functionally. From the experiments conducted in this study, it was shown that ineffectual instructions can be selectively removed from warm-up to reduce simulation times.

3.8 Appendix

Relative Error of all sampled experiments:

Relative Error										
	ammp	art	gcc	mcf	parser	perl	twolf	vortex	vpr	AVG
FP (20%)	0.0035	0.02	0.0032	0.0386	0.0402	0.0226	0.0094	0.0114	0.0162	0.0184
FP (40%)	0.0037	0.0037	0.0041	0.0364	0.021	0.0203	0.0082	0.0066	0.0164	0.0134
FP (80%)	0.0038	0.0009	0.006	0.0361	0.0101	0.0011	0.007	0.0039	0.0173	0.0096
None	0.0025	0.1665	0.2001	0.1472	0.4764	0.0897	0.4836	0.1795	0.3267	0.2302
$S_{\$}$	0.0202	0.0125	0.0397	0.0302	0.0625	0.0107	0.0517	0.0126	0.0347	0.0305
S_{BP}	0.0188	0.1574	0.1804	0.1525	0.4554	0.083	0.4648	0.1696	0.3176	0.2222
$S_{\$BP}$	0.0037	0.0009	0.0063	0.036	0.0085	0.0032	0.0054	0.0026	0.0168	0.0093
$R_{\$ (20\%)}$	0.0194	0.0118	0.0383	0.037	0.0919	0.0106	0.058	0.0028	0.0344	0.0338
$R_{\$ (40\%)}$	0.0197	0.0287	0.0392	0.0365	0.074	0.008	0.0574	0.008	0.0345	0.034
$R_{\$ (80\%)}$	0.0197	0.0319	0.041	0.0359	0.0644	0.0052	0.0561	0.0106	0.0357	0.0334
$R_{\$ (100\%)}$	0.0197	0.0321	0.0413	0.0356	0.0623	0.0105	0.0545	0.0118	0.0352	0.0337
R_{BP}	0.0186	0.1635	0.1907	0.1382	0.4597	0.0781	0.4691	0.1716	0.3189	0.2232
$R_{\$BP (20\%)}$	0.0029	0.0003	0.0055	0.0423	0.0429	0.0221	0.0123	0.0125	0.0169	0.0175
$R_{\$BP (40\%)}$	0.0033	0.0171	0.0067	0.0421	0.0224	0.019	0.0105	0.0072	0.0168	0.0161
$R_{\$BP (80\%)}$	0.0033	0.0203	0.0086	0.0414	0.0119	0.005	0.0095	0.0046	0.0178	0.0136
$R_{\$BP (100\%)}$	0.0033	0.0206	0.009	0.0411	0.0095	0.0002	0.0078	0.0035	0.0173	0.0125

Simulation time in seconds for all sampled experiments:

Time										
	ammp	art	gcc	mcf	parser	perl	twolf	vortex	vpr	AVG
FP (20%)	759.35	632	1336.1	2331.6	953.45	1004.1	911.33	809.54	653.31	1043.4
FP (40%)	934.81	780.96	1717.6	3046.3	1210	1256.4	1142	1060.5	862.36	1334.5
FP (80%)	1336.7	1030.3	2354.5	4012.7	1785.5	1762.8	1616.1	1436.3	1217.8	1839.2
None	548.4	523.65	913.86	1631.9	700.78	803.7	650.43	637.25	542.16	772.46
$S_{\$}$	1199.8	1016.5	1899.1	2773.4	1292.5	1428.8	1254.4	1188	936.63	1443.2
S_{BP}	945.11	646.44	1806.8	2435	1361.1	1302.3	1234.2	1012.4	926.27	1296.6
$S_{\$BP}$	1603.5	1181.4	1508.8	3235.8	3338.5	2038.3	1874.4	1662.2	1419.7	1984.7
$R_{\$ (20\%)}$	792.56	681.95	979.64	2664.4	1246.8	1064.6	892.36	804.75	643.2	1085.6
$R_{\$ (40\%)}$	896.78	765.98	1205.8	2830.1	1105.5	1115.6	983.88	955.88	731.06	1176.7
$R_{\$ (80\%)}$	1136.9	997.38	1285	3240	1428.9	1342.3	1134.8	1294.9	833.27	1410.4
$R_{\$ (100\%)}$	1244.5	925.12	1493.3	3429.2	1734	1535.9	1229.6	1276.1	919.3	1531.9
R_{BP}	650.84	505.1	800.77	2203.7	867.04	845.74	769.1	653.87	558.01	872.68
$R_{\$BP (20\%)}$	905.82	697.58	1076.4	3023.1	1360.5	1141.6	1018.4	930.69	735.63	1210
$R_{\$BP (40\%)}$	984.67	807.6	1196.6	3116.9	1461.6	1263.3	1122.1	1049.9	817.94	1313.4
$R_{\$BP (80\%)}$	1251	928.69	1428.6	3523.9	2068.9	1496.9	1270.5	1305.5	944.81	1579.9
$R_{\$BP (100\%)}$	1368.4	976.37	1544.5	3683.7	2396.4	1656	1346.2	1410.5	1003.8	1709.5

Results of constructed confidence intervals, which indicate whether or not a sampled experiment bracketed the true population IPC.

Confidence tests									
	ammp	art	gcc	mcf	parser	perl	twolf	vortex	vpr
FP (20%)	yes	yes	yes	yes	no	no	yes	yes	yes
FP (40%)	yes	yes	yes	yes	no	no	yes	yes	yes
FP (80%)	yes	yes	yes	yes	yes	yes	yes	yes	yes
None	yes	no	no	no	no	no	no	no	no
$S_{\$}$	yes	yes	no	yes	no	no	no	yes	no
S_{BP}	yes	no	no	no	no	no	no	no	no
$S_{\$BP}$	yes	yes	yes	yes	yes	yes	yes	yes	yes
$R_{\$ (20\%)}$	yes	yes	no	yes	no	no	no	yes	no
$R_{\$ (40\%)}$	yes	yes	no	yes	no	yes	no	yes	no
$R_{\$ (80\%)}$	yes	yes	no	yes	no	yes	no	yes	no
$R_{\$ (100\%)}$	yes	yes	no	yes	no	no	no	yes	no
R_{BP}	yes	no	no	yes	no	no	no	no	no
$R_{\$BP (20\%)}$	yes	yes	yes	yes	no	no	yes	yes	yes
$R_{\$BP (40\%)}$	yes	yes	yes	yes	no	no	yes	yes	yes
$R_{\$BP (80\%)}$	yes	yes	yes	yes	yes	yes	yes	yes	yes
$R_{\$BP (100\%)}$	yes	yes	yes	yes	yes	yes	yes	yes	yes

Accuracy and simulation time results for SimPoint experiments:

SimPoint Relative Error										
	ammp	art	mcf	gcc	parser	perl	twolf	vortex	vpr	AVG
50K	0.0215	0.0406	0.0923	0.2569	0.4103	0.2278	0.3408	0.1537	0.3262	0.2078
50K-SMARTS	0.2171	0.3206	0.0435	0.0235	0.0565	0.0226	0.0057	0.0636	0.0037	0.0841
10M	0.0485	0.003	0.0066	0.0246	0.0521	0.2308	0.0035	0.0117	0.0052	0.0429
10M-SMARTS	0.0485	0.008	0.0066	0.0193	0.1205	0.2303	0.0612	0.0121	0.0258	0.0591
SimPoint time										
	ammp	art	mcf	gcc	parser	perl	twolf	vortex	vpr	AVG
50K	501	856	850	1030	925	491	594	545	429	691.22
50K-SMARTS	1841	1119	3497	2576	3007	1451	1680	1561	1303	2003.9
10M	2686	1535	9389	2548	1444	979	669	1254	1026	2392.2
10M-SMARTS	3549	2179	12154	4421	3191	2205	1245	2279	1954	3686.3

CHAPTER 4

COMBINING CLUSTER SAMPLING WITH SINGLE PASS METHODS FOR EFFICIENT SAMPLING REGIMEN DESIGN

4.1 Statistical Sampling Assumptions

According to the central limit theorem, randomly extracted data from any non-normal distribution may be used to generate a normal distribution of sample means. From this normal distribution, conventional associations and formulas may then be applied to the normal distribution to find its mean, variance, etc. The inferences drawn from the normal distribution may then be associated with the non-normal distribution. When sampling from a large population of any distribution shape, the distribution of the sample means will approach the normal distribution with a sufficient sample size [36].

Two fundamental assumptions regarding sampling techniques are as follows: 1) increasing the sample size will increase the sample accuracy; and 2) the test for individual inclusion in the sample must be random, such that each element in the population must have the same probability for inclusion. A variety of factors affect sampling accuracy for a constant sample size, including the variance of the population and the type of distribution being sampled; however, as the sample size increases, accuracy also increases. Thus, with a sufficiently large sample size, the estimate of the mean approaches the true mean and the error approaches zero.

4.2 Sampling Regimen Construction

When designing a valid sampling regimen, three parameters must be determined: the cluster size, the number of clusters, and the location of the clusters within the dynamic instruction stream. The cluster size refers to the number of instructions executed in full cycle-accurate detail. The cluster number and size parameters dictate the sample size. Statistical simulation is a compromise between speed and accuracy. If too many clusters are selected, then speed will be sacrificed. If too few clusters are selected, then accuracy will be sacrificed. To further complicate this issue, the location of clusters is equally important. Even if the cluster number and size parameters are sufficient for sampling performance, the clusters may be located at non-representative sections of code and exhibit high sampling bias. Thus, measurements taken at such locations could lead to inaccurate estimates of performance. Many researchers have investigated techniques which reduce cold-start bias for microarchitectural simulation. However, little work has been proposed dealing with efficient techniques for sampling regimen design.

When designing a sampling regimen, each program-input pair may have dramatically different performance, affecting the underlying distribution of IPC. Thus, a sampling regimen that performs well for one workload will not necessarily be accurate when applied to other workloads (or even the same workload with different inputs).

Often, sampling regimen parameters are derived through an iterative process consisting of the following steps:

1) the workload under test is executed to completion for error comparison;
2) the workload is sampled via ad-hoc selection of the number of clusters and cluster size;

3) the results are analyzed to determine if confidence tests are met with sufficiently low error; and

4) if the error threshold or confidence tests are unsatisfactory, return to Step 2.

This iterative derivation of a valid sampling regimen can be extremely time-consuming. For example, consider a user who wishes to evaluate a number of sampling regimen configurations. Assume the user wishes to assess cluster sizes ranging from 1,000 to 50,000, with a step size of 1,000, and a cluster number ranging between 30 and 1,000. For simplicity, assume that each execution takes 30 minutes. Evaluation of this magnitude would include over 48,000 sampled simulations, and require over 2 years of simulation time. Even more time would be necessary if multiple random seeds were considered for each sampling regimen. It is unlikely an exhaustive regimen sweep such as this would be performed. However, the previous scenario represents an extreme example of the time-consuming nature of sampling regimen design.

Although a sampling regimen may be valid for a given workload, the user may be simply "unlucky" and randomly select non-representative locations. For example, a user could simulate the same sampling regimen 20 times in a row (utilizing different random seeds) and statistically, 1 could fail (a 95% confidence test implies that the true population mean has a 95% chance of being bracketed by the sample estimate). Therefore, a valid sampling regimen could be evaluated as a "false negative."

This work allows users to perform a single simulation to derive a valid sampling regimen and achieves the following goals: 1) prevent users from having to run the entire workload for performance comparison; 2) circumvent the iterative nature of regimen design; and 3) derive a valid sampling regimen according to user-specified criteria.

4.3 Single-Pass Regimen Design

Figure 15 shows a flow diagram of the single-pass regimen evaluation method. A program-input pair is first profiled via a very large sample. This large sample contains a very large number of clusters, many times more than are required for a valid sampling regimen. This sample is called the *over-sample* since the sampling rate is much higher than the minimum requirements. As the sample size increases, the estimate of the mean will converge to the true mean (according to the law of large numbers), and the over-sample is expected to accurately estimate the true mean. Embedded in this sample, information is contained regarding varying sized clusters.

After the over-sample has been collected, the estimate of the mean, or IPC, is assumed to be a highly accurate estimate of the true performance of the workload. The data collected in the over-sample were analyzed to generate a list of sampling regimen candidates. Given user-specified criteria, the candidate list is pruned. The final candidate list then identifies valid sampling regimen configurations. Reported data include the cluster size, cluster number, and starting location of each cluster.

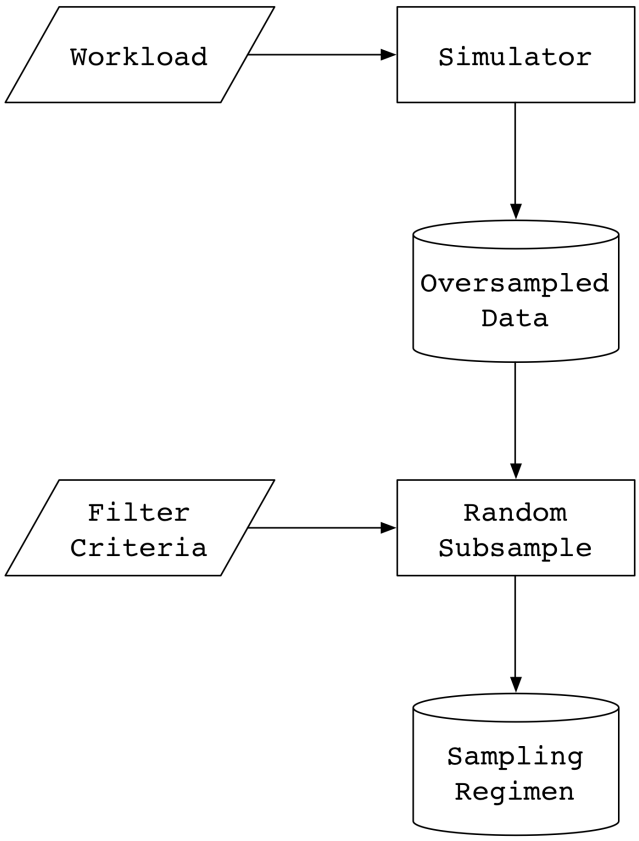


Figure 15: Single-Pass Sampling Regimen Design Flowchart

4.4 Simulator Modifications

Assuming a simulator that incorporates cluster sampling with SMARTS [84] warm-up, the modifications necessary to implement single-pass regimen design are minimal. This work was performed using SMARTS warm-up since it is generally accepted as the most accurate warm-up method. However, this method is orthogonal to warm-up, and could be applied using any warm-up method that effectively removes non-sampling bias.

The simulator was modified to ensure all aspects of the regimen configuration could be controlled; in this work a configuration file was used to specify regimen

attributes. Parameters could be specified via the command line, but the large numbers of inputs made this approach unattractive. The regimen configuration file was used to specify the size of a cluster, the number of clusters, and the starting location of each cluster within the dynamic instruction stream. Additional parameters for the profile sample included a minimum cluster size, a maximum cluster size, and a step size.

Execution of each cluster continued similarly as SMARTS with a few minor differences. As in SMARTS, warm-up was performed by functionally applying branch predictor and memory instructions during skipping. When the next cluster was reached, cycle-accurate simulation was performed, where each cluster was evaluated according to the specified minimum size. The difference between single-pass sampling regimen construction and SMARTS was that hot execution continued past the minimum cluster size, until the specified maximum cluster size had been reached. At each step size increment, all measured information in a cluster were saved. The recorded information allowed the user to determine which measurements would have been taken for each cluster size and location. Figure 16 shows a diagram of the accounting differences between normal SMARTS execution.

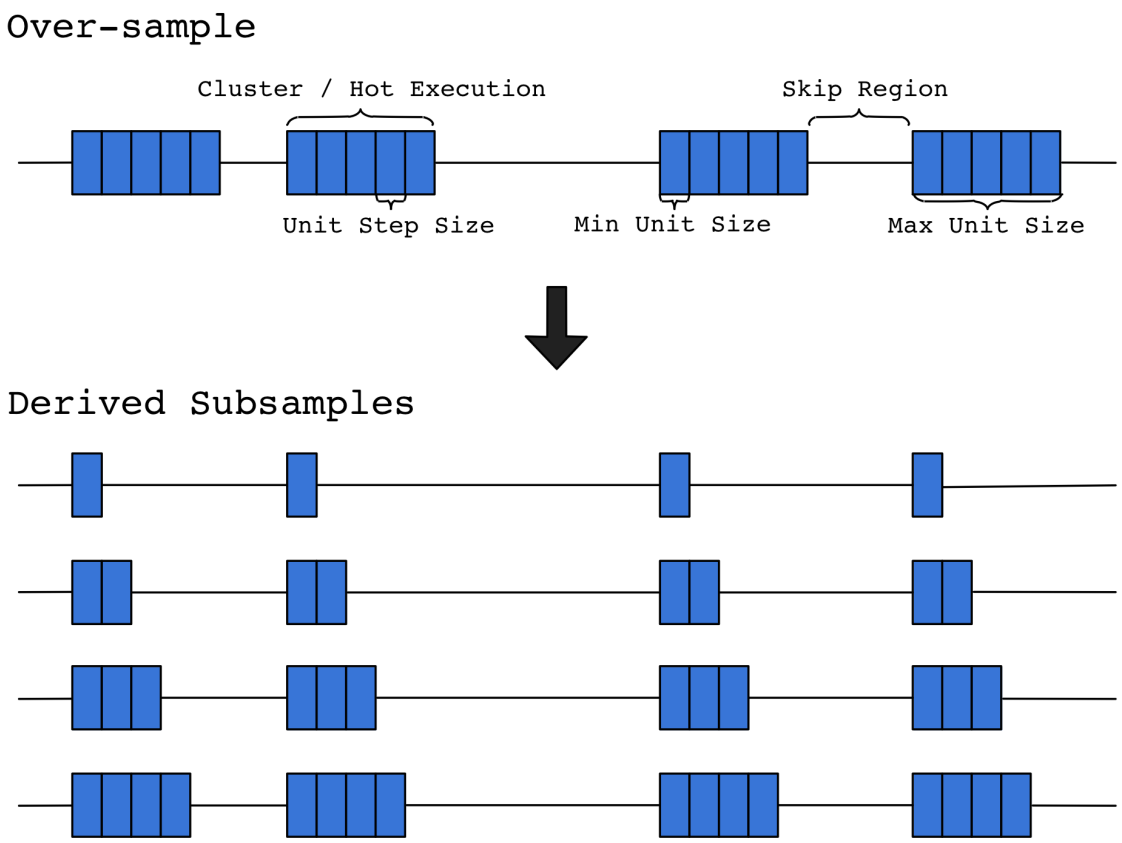


Figure 16: Cluster Sampling Modifications to Enable Derived Subsamples

4.5 Profiling Sample

Figure 17, Figure 18, and Figure 19 show the results of the over-sampled profile simulations. As expected, increasing the sample size increased accuracy, but also increased simulation time. Each workload was sampled such that a specified ratio of the entire workload was included into the over-sample simulation. Each workload was sampled according to the following sampled to non-sampled ratios: 1:6, 1:12, 1:24, 1:48, 1:96, 1:192, 1:384, 1:768, 1:1536, and 1:3072. The 1:6 sample ratio indicated that 1 out of every 6 instructions in the entire workload was simulated in full detail.

Figure 17 shows the sampling accuracy associated with each sampling ratio. As expected, the smallest sampling ratio, 1:3072, (corresponding to 0.03% of the workload) had the highest error. For this ratio, *gcc* had the highest error with 24%, and *art* had the lowest error with 0.4%. As the sampling ratios increased, the error rates for all benchmarks decreased. At the highest sampling ratio, the average relative error was 0.03%. Figure 18 shows a magnified version of Figure 17 at the largest two sample sizes. As expected, the highest sampling ratio of 1:6 achieved the lowest error (0.3% on average). At a 1:6 sampling ratio, *ammp* had the highest relative error at 1.7%.

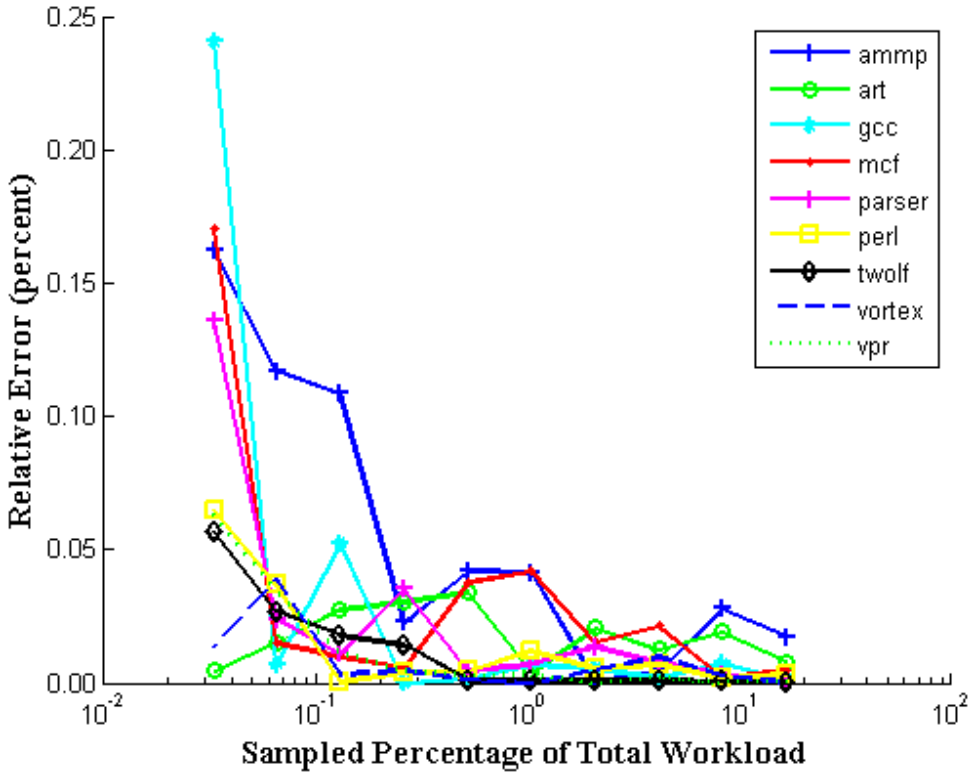


Figure 17: Single Pass Error vs. Sample Size

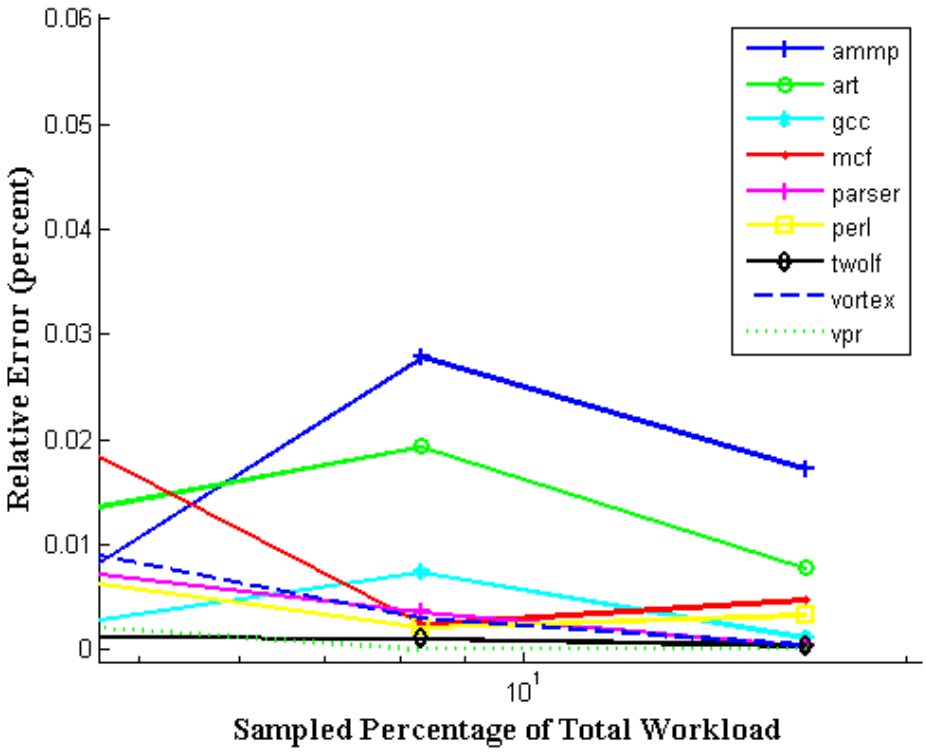


Figure 18: Single Pass Error Rates at High Sampling Rates

As shown in Figure 19, the simulation times required for over-samples increased exponentially (along with the exponential increase in sample sizes) for certain benchmarks. At the highest sampling ratio, 1:6, the average execution time was approximately 5.4 hours. *Mcf* took the longest at this ratio at 22.8 hours. However, the largest sampled ratio was not necessary to obtain highly accurate IPC estimates. Approximate error rates were also observed when a sampling ratio of 1:48 was used. At this sampling ratio, the average relative error was 0.7% with an average execution time of 1.5 hours. Although convergence occurred at different sampling ratios for each workload, all experiments obtained accurate results when a sampling ratio of 1:48 was used.

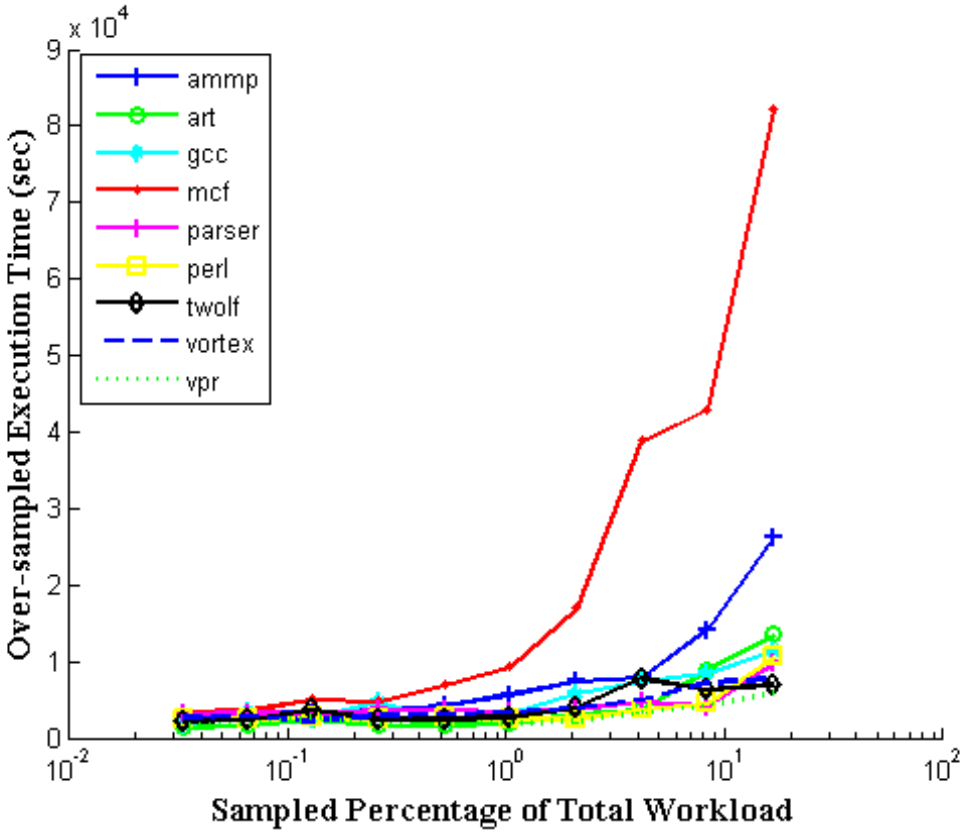


Figure 19: Single Pass Sample Size vs. Execution Time

4.6 Profiling Analysis

After the over-sampled data were collected, their results were analyzed. Using this information, random subsamples were compiled over all ranges of possible cluster sizes and numbers of clusters. For each cluster size and cluster number, elements were randomly extracted from the over-sampled population. Each over-sampled subset was evaluated in terms of relative error, variance, and statistical confidence. Statistical confidence checks were used to ensure that random subsamples correctly bracketed the over-sample estimate (as well as the true population mean).

As previously stated, it is possible for a valid sampling regimen configuration to be rejected if non-representative elements from the population were included in the sample. If sampling regimen decisions were made solely upon the performance of a single sample, the presence of sampling bias may result in an incorrect classification (e.g., sufficient or insufficient). To reduce this possibility, each sampling regimen under consideration was sampled multiple times. From these results, the average expected level of performance for a given sampling regimen was obtained. Each sampling regimen was evaluated multiple times with different random seeds. In this work, each sampling regimen was tested 30 times in correspondence with the central limit theorem. Once all possible input combinations of cluster sizes and the number of clusters were searched, the algorithm proceeded to candidate selection.

The benefit of this profile analysis is twofold. First, the total workload simulation was not required since the over-sampled estimate of the mean was assumed to have sufficiently converged towards the true population mean. Second, all cluster sizes and numbers of clusters recorded in the over-sampled population could be simultaneously evaluated for inclusion.

The single-pass sampling regimen method of candidate analysis was statistically valid since each element in the over-sampled population had an equal probability of selection. Each sampling regimen was evaluated through repeated trials, and each sample was based upon the technique of uniform random selection. Furthermore, the use of over-sampled data provided confidence bounds that bracketed the true performance of each workload.

4.7 Candidate Selection

After the over-sample profile was analyzed and a list of candidates generated, a sampling regimen configuration was selected. From the list of potential sampling regimen candidates, candidates were pruned according to user-specified criteria. In this study, elements from the candidate list were conservatively pruned if all trials did not pass confidence tests relative to the over-sampled estimate. Furthermore, the minimum error could not exceed 2% and the variance could not fall below 0.02. These threshold values were arbitrary, and could be tailored according to user specified characteristics.

A small amount of sampling error was expected to be present when comparing the full execution of a workload to the over-sample profile. Additionally, sampling error was also expected to be present when comparing the second level simulation, provided by candidate selection, with the over-sample. By restricting the minimum error to 2%, the total bias introduced by sampling was kept sufficiently low to bracket the true performance of the workload. After the candidates were pruned, they were sorted based on sample size, which must be equal to the cluster size multiplied by the number of clusters. Candidates were sorted in this manner to reduce the overall execution time. For example, assume two sampling regimen configurations passed the user-specified criteria for candidacy. Given that one may require 2000 clusters with a cluster size of 50,000 instructions, and another may require 40 clusters with a size of 1,000 instructions, the latter should be chosen since it would require significantly less time to simulate.

The process of pruning sampling regimen candidates did not prevent cluster inclusion within future sampling regimen configurations. The decision to prune a

sampling regimen simply means that alternate cluster size and cluster number combinations should be explored.

4.8 Methodology

All experiments were conducted with the SPEC2K benchmark suite. Integer benchmarks used included *gcc*, *mcf*, *parser*, *perl*, *twolf, vortex*, and *vpr*. Floating point benchmarks used included *ammp* and *art*. The first six billion instructions from each benchmark were simulated using reference input sets.

The model used in this study was an execution-driven simulator based on SimpleScalar [14]. The processor pipeline could fetch and dispatch eight instructions per cycle, and could issue and retire four instructions per cycle. The model included eight universal function units, which were fully pipelined. The maximum number of in-flight instructions was 64. The issue queue size was 32, and there was a load-store queue of 64 elements. The pipeline depth was seven stages. The minimum branch miss-prediction penalty was five cycles. The processor frequency was assumed to be 2 GHz. The branch predictor was a 64K entry Gshare with an eight-entry return address stack. The BTB consisted of 4K entries. Architectural checkpoints were utilized to allow the processor to speculatively execute beyond eight branches.

A substantive memory hierarchy was modeled within the simulator. The first level data cache was 4-way and contained 32 KB with a 64-byte line size. The first level instruction cache was also 4-way and contained 64 KB with a 64-byte line size. The instruction and data caches were implemented using a write-through no-write allocate policy. The second level cache was 8-way and contained 1 MB with a 64-byte line size, and was implemented using a write-back write-allocate policy.

A bus model also was incorporated in order to emulate arbitration, contention, and transfer delay between the levels of memory. The first level bus was shared between the first level data and instruction caches, and connected the first level caches to the second level cache. The first level bus had a width of 16 bytes and operated at 1 GHz. The second level bus connected the second level cache to main memory, had a width of 32 bytes, and operated at 2 GHz.

The model included both a functional and a timing simulator. The functional simulator was used to validate the results of the timing simulator. If the timing simulator attempted to commit a wrong value, the functional simulator would assert an error. However, in the context of sampled simulation, the functional simulator had additional uses. As instructions in the dynamic stream were skipped (either in cold or warm simulation), the functional simulator retained valid architectural state. When hot execution continued in the next cluster, the values of the registers contained in the functional simulator were copied to the timing simulator. For processor simulations, the measured performance metric was IPC, which was defined as the number of instructions retired per execution cycle.

4.9 Results

Figure 20 shows the reduction in simulation time of single-pass simulation compared to the total workload simulations as a factor speedup. As expected, 16.67%, or a sampling ratio of 1:6, had the least savings due to the size of the over-sample. All results in Figure 20 calculated speedup including simulation times of both the over-sample and the second level simulation indicated by the regimen candidacy selection, compared to the full (un-sampled) simulation.

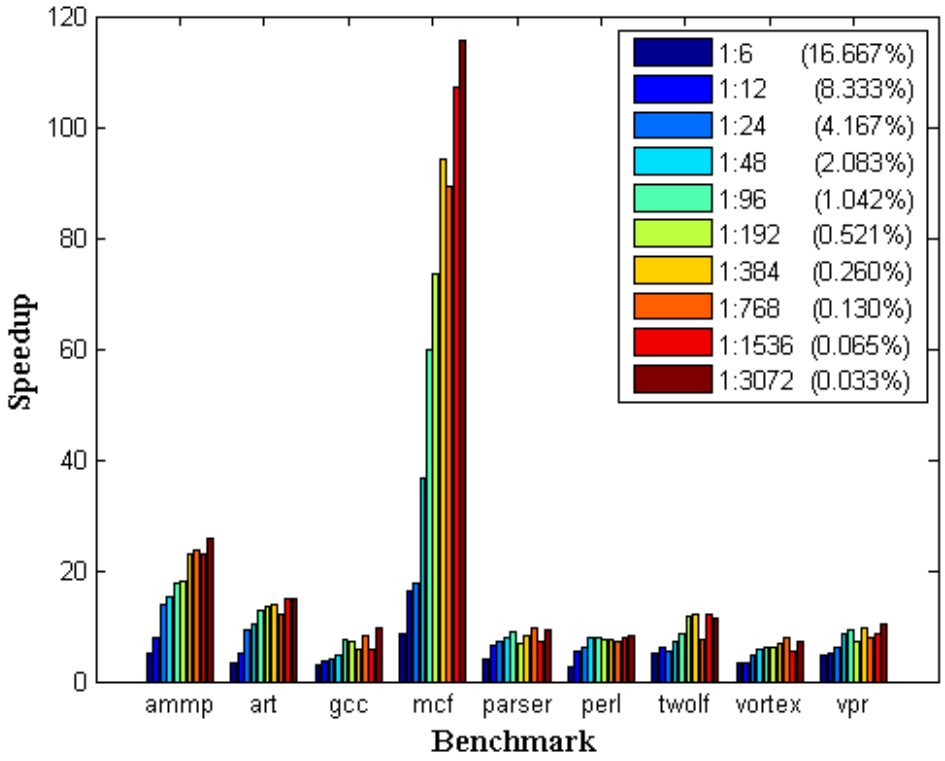


Figure 20: Single-Pass Sampling Regimen Simulation Speedup

As previously mentioned, there were two types of error which were introduced in the single-pass sampling methodology: 1) the error between the profile sample and the full execution; and 2) the error between the single-pass simulation (second level simulation) and the profile sample. All second level simulations passed confidence tests when compared to the profile sample. At sampling ratios of 1:6, 1:12, 1:24, 1:48, 1:96, and 1:192, all second level simulations passed confidence tests when compared to the full execution.

Low sampling ratio simulations passed most of the confidence tests when compared to the total execution. At sampling ratios of 1:384, 1:768, and 1:1536, 8 of 9

workloads passed confidence tests. At sampling ratios of 1:3072, 6 of 9 workloads passed confidence tests. This was expected behavior since error was expected to increase as the sample size decreased.

The lowest sampling ratio exhibited the highest factor speedup, with an average of 23.6x faster, where *mcf* was 115x faster than the entire workload simulation. When the sampling ratio of 1:192 was utilized a factor speedup of 16.85x was obtained. In this study, a sampling ratio of 1:192 was optimal in achieving high speed without sacrificing accuracy (less than 1% error). Using this sampling ratio, the average number of clusters was 130, and the average cluster size was approximately 5,000 instructions.

4.10 Related Work

When sampling a workload, numerous works have been proposed to study effective techniques for non-sampling bias removal [13], [23], [28], [29], [30], [33], [34], [82], [83], [84]. Non-sampling bias removal is paramount before a sampling regimen may be constructed. Without warming processor state, measurements taken from the individual sampling units could be inaccurate. Thus, the construction of a sampling regimen may only be addressed after non-sampling bias has been removed.

The construction of a valid sampling regimen refers to the selection of sampling units that are representative of the overall population distribution. Currently, most techniques which aid in the construction of a sampling regimen are focused on stratified sampling techniques [47], [56], [70], [73], [75]. Liu, et al. [56] used profiling to partition an application into a number of static code sections. Characterization was performed on the different strata to determine the degree of sampling. Srinivasan, et al. [75] used a phase detection algorithm based on IPC traces to partition the workload into strata (or

phases). Elements were sampled from each strata and the distribution of the sampled elements was compared against the distribution of the strata using a proposed chi-squared similarity measure.

SimPoint [70], [73] is another stratified sampling technique that used basic block vector (BBV) profiling in order to identify representative instruction windows from the dynamic instruction stream. The entire workload is first profiled in functional simulation in order to count the frequently executed basic blocks at a specified window size. A k-means clustering algorithm is used to determine the instruction windows that are to be executed by the user. A prerequisite of this algorithm requires the user to specify a maximum number of windows to be executed (maxK). The goal of SimPoint is to locate one or more instruction windows to be executed that represent the execution of the entire workload. Many architects use SimPoint to identify a single instruction window for simulation. However, a single instruction window (corresponding to a sample consisting of a single sampling unit) does not translate into a valid sampling scheme. Furthermore, reliability metrics such as the confidence interval cannot be applied to samples that consist of only one sampling unit. Since its inception, SimPoint has been extended to Variance SimPoint [70], which uses a parametric bootstrap technique in order to estimate confidence intervals for the strata identified through the BBV vectors. Using this technique, profiling analysis is performed on randomly chosen samples from each of the clusters.

In cluster sampling techniques, the design of a sampling regimen has typically been iterative in nature [23], [84]. In order to evaluate each sampling regimen, the entire workloads of interest are typically executed in their entirety. Results for a specific

sampling regimen are compared against the full workload and repeated until the error has been sufficiently reduced. The iterative nature of sampling regimen design can be very costly and time-consuming -- both in the execution of the entire workload, and in the iterative selection of sampling units.

4.11 Conclusion

In this work, an efficient sampling regimen design process was presented. Utilizing a profile sample, thousands of sampling regimen configurations could be simultaneously evaluated for accuracy and statistical confidence. Each sampling regimen configuration was tested multiple times, each yielding a different random sample in order to increase the probability of correct classification. From the profile analysis, a list of possible sampling regimen configurations were identified and then pruned according to user-determined filter criteria. From this candidate list, sampling regimens were sorted based on sample size to allow the user to collect the smallest, and therefore fastest, sample.

The techniques presented in this study are a vast improvement over traditional sampling regimen design. In this work, it was shown how sampling regimens can be designed without requiring the entire workload to be simulated for accuracy comparisons. Significant time-savings were realized since full workload simulations were not required. Additionally, the efficient single-pass sampling regimen design algorithm resulted in significantly faster simulation.

CHAPTER 5

CURRENT OBSTACLES THAT PREVENT ACCURATE AND RELIABLE MULTI-THREADED SAMPLING

Contemporary physical constraints, most notably the power wall, have necessitated a paradigm shift in microarchitectural design. Whereas multi-core systems currently contain a handful of cores, it is expected future designs may contain hundreds or even thousands of cores on a single die, ushering in the era of manycores. In order for such systems to become a reality, the industrial and academic communities must first tackle a number of challenges, including: determining the fundamental hardware building blocks (and how to interconnect them) as well as providing new programming models to effectively and efficiently use system resources [5]. In prototyping potential solutions to these problems, detailed time-step simulation is vital for exploring the design space of potential architectures.

Although the shift to manycore systems has allowed designs to continue in accordance with Moore's Law, this trend has created challenges for both developers and computer architects alike. The parallelization of many serial programs is often nontrivial and may require different algorithms requiring software systems to be completely redesigned. Therefore, developers wishing to harness the full power of their machines must write programs that consider parallel execution from the very beginning of the design process. Computer architects also face the challenges of designing systems that grow in complexity in proportion with the number of processors. Notably, cache

coherency becomes increasingly complex at higher core counts since simpler broadcast-based solutions (e.g., snoopy bus) do not effectively scale.

As more processors are added to simulated environments, the simulation times required increase dramatically when the simulator is single-threaded. Not only does the complexity of the system increase with the number of processors, but the number of instructions to be simulated also increases as well (and may even increase superlinearly). During the execution of a parallel program, instructions are divided among the available processors. The division of work among the processors must be done carefully, and introduces additional accounting instructions[11] to coordinate execution. These instructions introduced through parallelization are non-useful, since they do not contribute to the execution of the program and yet they are required to execute the program correctly. In shared memory systems, accounting instructions typically arise from the execution of mutexes or other synchronization primitives used to guard against race conditions. If the overheads arising from these extra instructions are small, then native execution may approach the theoretical speedup over the single-threaded execution. If the overheads are large, then native execution times may be substantially degraded (and could even result in an execution slower than single-threaded execution). In either case, the amount of work which must be performed by the simulator increases superlinearly with the number of processors since more work must be performed to execute the program. Although the simulator itself may be parallelized to overcome the massive slowdowns incurred by simulating more processors, numerous contemporary

[11] This assumes the execution of multi-threaded programs on shared memory systems. However, distributed memory systems would also involve instruction overheads associated with ushering data to and from processors' private memory spaces.

manycore simulators remain single-threaded [7], [9], [60], [72]. As future designs increase in core counts, it is expected the gap between native and simulated execution speeds will increase, further compounding the problems already evident in simulation speeds.

Long simulation times have been and remain to be one of the primary bottlenecks for practicing architects. To address these issues, numerous strategies to accelerate simulation have been proposed. Previously proposed single-core solutions have included reducing the simulation effort by reducing the workload (e.g., reduced input sets [46], statistically synthesized workloads [1], statistically sampled simulation [12], [13], [29], [50], [77], [84], SimPoints [70] and benchmark subsetting [40]), optimizing simulation tasks (e.g., direct execution [24]), and parallelization of the simulator [31], [50]. Unfortunately, none of these acceleration techniques can be applied to the simulation of multi-threaded workloads[12].

5.1 Multi-threaded Sampling Obstacles

Sampled simulation can dramatically reduce the effort required to obtain accurate performance estimates for single-threaded, multiprogrammed or throughput-oriented workloads. Rather than simulating the entire workload, *sampling* allows small fractions of the workload to be simulated in full detail. Measurements (i.e., *clusters*) are obtained from sampling units, where functional fast-forwarding is used to skip between measurement sites, and the sampled measurements are used to obtain point estimates.

---

[12] One exception is [8], which estimates cache miss rates of multi-threaded workloads. As will be discussed in this chapter, cache miss rates cannot be used to estimate execution time of a multi-threaded program (or speedup) reliably.

Additionally, statistical confidence interval calculations may be used to assess the reliability of estimated performance based solely on sample data. The accuracy of sample measurements is improved by using warm-up methods to reduce the non-sampling bias associated with skipping to arbitrary points within the program. For single threaded simulation, there exist multiple techniques to accurately reconstruct simulator state [13], [29], [79], [82], [84], [85]. For multi-threaded workloads, warm-up has been applied to reconstruct directory and cache state in a multiprocessor simulation [8].

The term *multi-threaded workload* is somewhat opaque, since it may refer to many different types of programs, and by extension, simulation environments. To clarify, the rest of this dissertation refers to multi-threaded workloads as *single-application, multi-threaded workloads* executing on manycore systems. Most techniques that have been used to accelerate multi-threaded workloads focus on multiprogrammed workloads (i.e., multiple independently running programs), or transaction-oriented workloads (i.e., largely independent transactions interacting with a common server). However, single-application, multi-threaded workloads generally have higher degrees of inter-thread communication and inter-thread dependence, rendering many previously proposed acceleration techniques ineffective. This chapter describes the challenges associated with sampling multi-threaded programs, and the reasons previously developed methodologies cannot be applied in their current form. Furthermore, this chapter defines ***Thread Skew***, a metric used to measure the differences in thread progressions between an un-sampled (fully simulated) program and a sampled one.

5.1.1 Theoretical Sampling Extensions to Multi-threaded Applications

To effectively explain the reasons sampling cannot be easily applied to a multi-threaded programs, an assumed sampling procedure is first described so the challenges may be enumerated. The phases of the proposed sampling methodology are similar to those in single-threaded sampling, and consist of warm, cold, and hot phases. To reduce sampling bias, sampling is performed over both the serial and parallel regions of execution. During serial code regions, sampling may be performed identically to single-threaded sampling. During parallel code regions, sampling units are randomly taken for all running threads. Sampling units are then combined to form a population estimator of the desired characteristic. As will be discussed later in this chapter, issues with multi-threaded sampling begin to emerge at the very beginning (i.e., the selection of a metric to estimate).

Figure 21 shows a theoretical extension of single-threaded sampling procedures for multi-threaded workloads. The fork-join model of parallelism is common to many programming frameworks (OpenMP, CUDA, POSIX Threads, etc.), and describes an important class of computation. The proposed sampling framework assumes the number of threads is equal to or less than the number of processors to avoid non-sampling bias arising from thread preemption.

Clusters are obtained from both serial and parallel code sections. During parallel execution, the main thread spawns worker threads, which perform some computation and then wait on a barrier (i.e., join) before serial execution resumes. During parallel execution, threads may enter and exit critical sections guarded by synchronization constructs such as condition variables, mutexes, semaphores, and

condition variables. The program may transition between multiple phases of parallel and sequential execution. After measurements are taken from all running threads, the program is fast-forwarded by randomly skipping instructions for all *running* threads (since blocked or idle threads cannot progress). The correct fast-forwarding of threads must successfully predict complex thread interactions. For example, during fast-forwarding a thread in the running state may transition to the to the blocked state due to suspension. Conversely, a thread in the blocked state may transition to the running state if a guarded resource is released. To guarantee the reduction of non-sampling bias introduced by divergent thread schedules, functional fast-forwarding must preserve the following two properties with respect to the un-sampled program: 1) threads must enter critical sections in the same order; and 2) the relative starting positions of all threads must be consistent. Ensuring the correct starting positions of all threads at the beginning of a sampling unit dictates the performance of each system thread must be accurately predicted at all points during functional fast-forwarding. Without knowing each threads' progress during fast-forwarding, threads may enter critical sections in a different order, thereby affecting the progression of other threads. The failure to preserve both properties could result in executions that differ dramatically from the un-sampled execution, resulting in high non-sampling bias and erroneous sampled measurements[13]. If the number of threads was allowed to be larger than the number of available cores, then the above conditions must be preserved, along with the thread priorities and subsequent thread schedules at every time slice.

---

[13] The relaxation (i.e., approximation) of one, or both, properties may be accomplished only through rigorous characterization of their effects upon non-sampling bias.

However, the challenges towards multi-threaded sampling are more fundamental than non-sampling bias components alone. Even if all of the previously stated issues with non-sampling bias were resolved, multi-threaded sampling would still suffer from the lack of representative performance metrics.

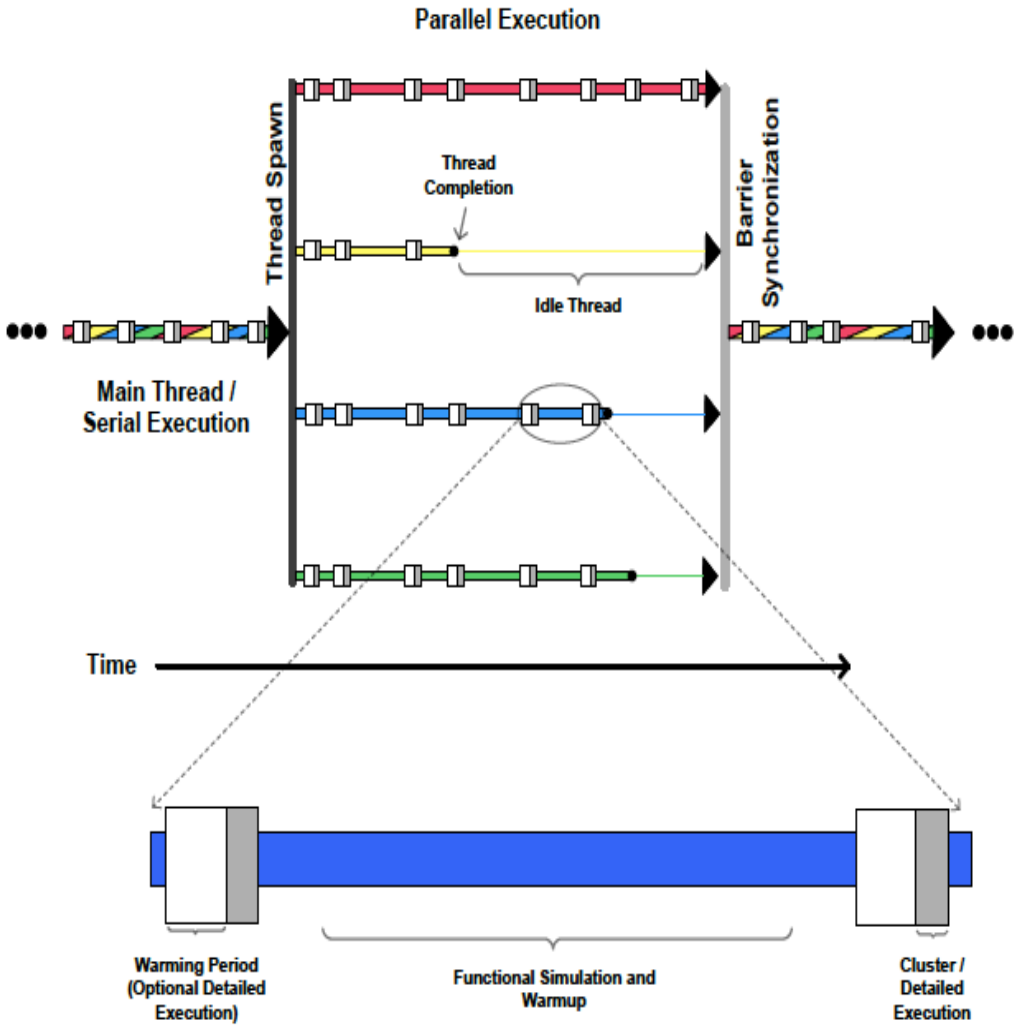


Figure 21: Theoretical Extension of Single-Threaded Sampling to Multi-threaded Workloads

5.1.2 The Lack of Stable Performance Metrics

A metric is considered *stable* if it can correctly track system trends as designs under test are modified. In other words, a stable metric is representative of program behavior if it can be used to reliably predict whether or not an architectural change is beneficial or detrimental towards system performance. The introduction of manycore processors has introduced additional challenges in the identification of stable performance metrics. For single-threaded workloads, IPC has historically been a popular metric to track system performance. However, the use of IPC as a metric for manycore environments may not be representative, and in some cases wildly inaccurate. In [2], Alameldeen and Wood discuss issues related to the use of IPC for multiprocessor workloads, and evaluate various microarchitectural changes to show that inaccurate conclusions could be drawn based upon IPC measurements alone. For example, in [2], IPC measurements incorrectly indicated a stride-based prefetcher degraded performance over a baseline system without a prefetcher.

The total execution effort of a parallel program is a combination of useful work, parallelization overheads, and idle times. For multi-threaded, parallel workloads, IPC does not distinguish between useful and non-useful work. For example, while executing a spin-lock loop, a thread waiting to acquire a lock typically executes in a tight loop (before yielding and suspending its execution) and will have extremely high IPC, even though the thread is not executing any instructions that contribute to useful work. Comparisons based on IPC also assume the number of instructions is fixed across executions. In manycore systems, instruction counts may vary due to mutex races can create divergent OS schedules. This effect becomes more severe as the number of cores

increases. As a result, errors associated with speedup computed from IPC measurements increase with the number of processors.

Researchers have proposed solutions to mitigate problems associated with IPC, which include: ignoring system code, ignoring thread idle times, exclusion of spin-lock loops and atomic operations, and using trace-drive simulation (or execution-driven simulation without non-determinism) [54]. As stated in [2], proposals to salvage IPC in multiprocessor systems are incomplete solutions. Ignoring system code is not effective for many commercial benchmarks (e.g., OLTP), which spend consider portions of their execution in system code. The exclusion of synchronization behaviors and thread idle times are also problematic, since they ignore non-useful work portions of the execution. If the contributions of non-useful work upon instruction counts are small (as was observed in [2]), then its exclusion is unlikely to stabilize IPC. If the contributions of non-useful work upon instruction counts are large, then non-useful work is contributing to significant portions of execution that must be taken into consideration. Since runtime is a function of both useful and non-useful work, the exclusion of non-useful work could lead to the overestimation of system performance.

The elimination of IPC as a viable metric is detrimental towards sampling manycore systems. Single-threaded, sampled simulation studies have typically relied upon IPC (and CPI by corollary) to estimate system performance. Although other rate-based metrics such as cache miss rates may potentially be estimated, they cannot be trusted to predict system performance. Without a *rate-based, stable performance metric*, measurements obtained through cluster sampling are not meaningful, since sampling theory relies upon rate-based metrics for the estimation of population characteristics.

Parametric and non-parametric statistics rely either directly, or indirectly, upon estimates of the mean (rate-based values) to estimate the population mean (a rate-based value).

In manycore systems, one metric commonly used to evaluate the effectiveness of a design is speedup, computed as a ratio of execution times. If the execution time is used for design comparison, then sampling cannot be applied. **Unlike rate-based metrics (such as IPC), the execution time of a program contains no parent distribution from which to sample – it is simply a static value.** In other words, the execution time of a program is a scalar quantity that sampling cannot estimate. **Thus, even if all other non-sampling bias components of manycore simulation were reduced, sampling cannot be applied until a stable rate-based performance metric is identified.** Furthermore, speedup cannot be estimated through sampling until reliable methods are discovered to accurately approximate the execution time (and ratios of execution times) through rate-based metrics.[14]

5.1.3 The Circular Dependence Dilemma of Parallel Workload Simulation

Single-threaded sampling mechanisms generally rely upon the application of warm-up methods to obtain accurate sampled estimates. In single-threaded sampling, the initial state that must be reconstructed for accurate measurement is typically limited to cache contents and branch predictor state.

In multi-threaded sampling, the initial state would include not only the cache tag-store and branch predictor state, but also the coherence state of cache blocks, and directory state including presence bits and directory coherence state. The reconstruction

[14] Even if these issues are overcome, it is unclear how to calculate confidence intervals from a ratio computed from two sample means.

of cache and directory state much be performed carefully to ensure deadlocks are not introduced. For example, consider the following hypothetical functional warming scenario. A directory is incorrectly warmed such that a cache block in the owned state is incorrectly associated with a particular cache. During cluster execution, a GETX request is issued by another processor, which forwards the request to the appropriate directory. The directory will, in turn, update its state to reflect that the requesting processor is now the owner and forward the request to the appropriate cache to provide the requestor with the necessary data. When the message arrives at its destination, the cache block will not be resident and the cache will not be able to respond, resulting in a deadlock. In single-threaded sampling, errors during warm-up could result in measurements that differ from the true execution, but the simulation will still be able to continue. In multi-threaded sampling, errors during warm-up could result in situations that prevent further simulation.

In multiprocessor systems, performance is a combination of individual thread executions, which depend upon and are affected by system state. Thread interactions occur implicitly through shared resources (e.g., a shared LLC) or explicitly through synchronization constructs. Race conditions due to resource locking may not be predictably modeled unless detailed state information regarding cache contents, system coherence state, core proximity to the home node, network contention, etc., are known. For example, consider the common practice of skipping initialization code at the beginning of a workload. For uniprocessor systems, solutions to the *cold-start problem* have been extensively studied [12], [13], [29], [85]. In multiprocessor systems, previously studied solutions are limited to fast-forwarding over serial code regions. If fast-forwarding terminates in a region of parallel thread execution, not only is system

state unknown, *but the relative thread progression and thread interleavings are unknown as well*. Effectively compensating for cold-start involves reconstructing system state, and requires precise knowledge of each individual thread's progress. However, the reconstruction of each thread's progress requires knowledge of system state to determine, for instance, the order that threads acquire and release critical sections. The approximation of system state, therefore, is dependent upon individual thread progressions, and the approximation of thread progressions are dependent upon system state, resulting in a circular dependence dilemma.

5.1.4 Relative Thread Progression and Thread Skew

In order to measure thread divergence quantitatively, and thus the impact of the circular dependence dilemma, the *thread skew* metric was developed. *Thread Skew* measures the divergence of thread progressions (non-sampling bias) between two simulations: one simulation that uses functional fast-forwarding (where thread divergence is introduced through imprecise skipping), and another that performs full-simulation from the beginning of the program. A formal definition of thread skew is shown in Figure 22.

Skew values are measured at the beginning of various program locations. For each location with imprecise skipping, the fetch counts[15] of all threads are summed to obtain a total system fetch count. Full simulations are performed to profile the fetch counts of all threads when the system arrives at the same total system fetch count. The total fetched instruction count provides a system-wide estimator of progress that is used to map divergent executions between the two simulations. Since the full-simulation

[15] The fetch counts used in the calculation of the thread skew metric exclude instructions that occur within thread synchronization functions.

outputs thread progress at a system fetch count dictated by the sampled simulation, the degrees of freedom of the thread skew measurement are reduced. In addition, comparing thread progressions at a constant system fetch count causes the skew values for all threads to sum to zero (at each cluster), since for every thread that leads true execution, another must lag. Threads leading true execution (full-execution fetch counts) have positive thread skew, and those lagging have negative thread skew.

For each cluster, *C*,:

1) From a sampled simulation, at the beginning of a cluster:

- *Record* sys_fetch_C *and all* t_i

where,

$t_{C,i} = fetch\ count\ of\ thread\ i$

$N =$ # threads

$sys\_fetch_C = \sum_{i=1}^{N} t_{C,i}$

2) From a full simulation

$t'_i = fetch\ count\ of\ thread\ i$

$sys\_fetch'_C = \sum_{i=1}^{N} t'_i$

when (sys_fetch_C' == sys_fetch_C):

$t'_{C,i} = t'_i\ \forall i$

3) Calculate thread skew from profiled data

$thread\_skew_{C,i} = t_{C,i} - t'_{C,i}$

When $sys\_fetch'_C = sys\_fetch_C$

$\sum_{i=1}^{N} t'_{C,i} = \sum_{i=1}^{N} t_{C,i}$

$\sum_{i=1}^{N} t_{C,i} - \sum_{i=1}^{N} t'_{C,i} = 0$

$\sum_{i=1}^{N} (t_{C,i} - t'_{C,i}) = 0$

$\sum_{i=1}^{N} thread\_skew_{C,i} = 0$

Furthermore, at barrier releases:

$thread\_skew_i = 0$, for all i

Figure 22: Formal Definition of Thread Skew

Thread skew can be classified into two categories: *asynchronous* and *synchronous*. *Asynchronous Thread Skew* (ATS) arises from non-uniform thread progress of functional skipping between sampling units, due to a combination of varying IPC and thread scheduling. Non-sampling bias effects introduced into the system by

asynchronous thread skew are applicable directly to transaction oriented computing, and are discussed in [45]. In [8], thread skew was mitigated by repeatedly sampling the workload to probabilistically bound thread progressions. Synchronization events (i.e., mutexes, barriers, semaphores, or condition variables) cause variation in thread orderings and lead to uncertainty in thread state after functional skipping, which are referred to as *Synchronous Thread Skew* (STS). STS is due to the phenomenon where, as is previously discussed, if many competing threads are awaiting access to a lock, then the ordering of thread executions within the critical section is dependent on coherence state, physical proximity to the home node, and network contention. At the beginning of program execution, threads are perfectly aligned to their true execution, but begin to diverge from their true alignment quickly. During simulation, absolute thread skew increases for many threads until a barrier is encountered. Barriers synchronize all threads to a fixed program location and collapse thread skew to zero. The absolute value of the sum of thread skew tends to increase after a barrier release until another barrier is encountered. The method of barrier-interval simulation (see Chapter 6) avoids the problem of thread skew, since intervals begin immediately following barrier releases where thread skew is guaranteed to be zero. Thread skew is shown graphically for five *SPLASH-2* workloads at varying core counts (i.e., 2, 4, 8, 16, 32, 64, 128, 256, and 512 cores). Figure 23 shows thread skew for *FFT*. Figure 24 shows thread skew for *LU contiguous*. Figure 25 shows thread skew for *OCEAN contiguous*. Figure 26 shows thread skew for *RADIX*. Figure 27 shows thread skew for *WATER SPATIAL*.

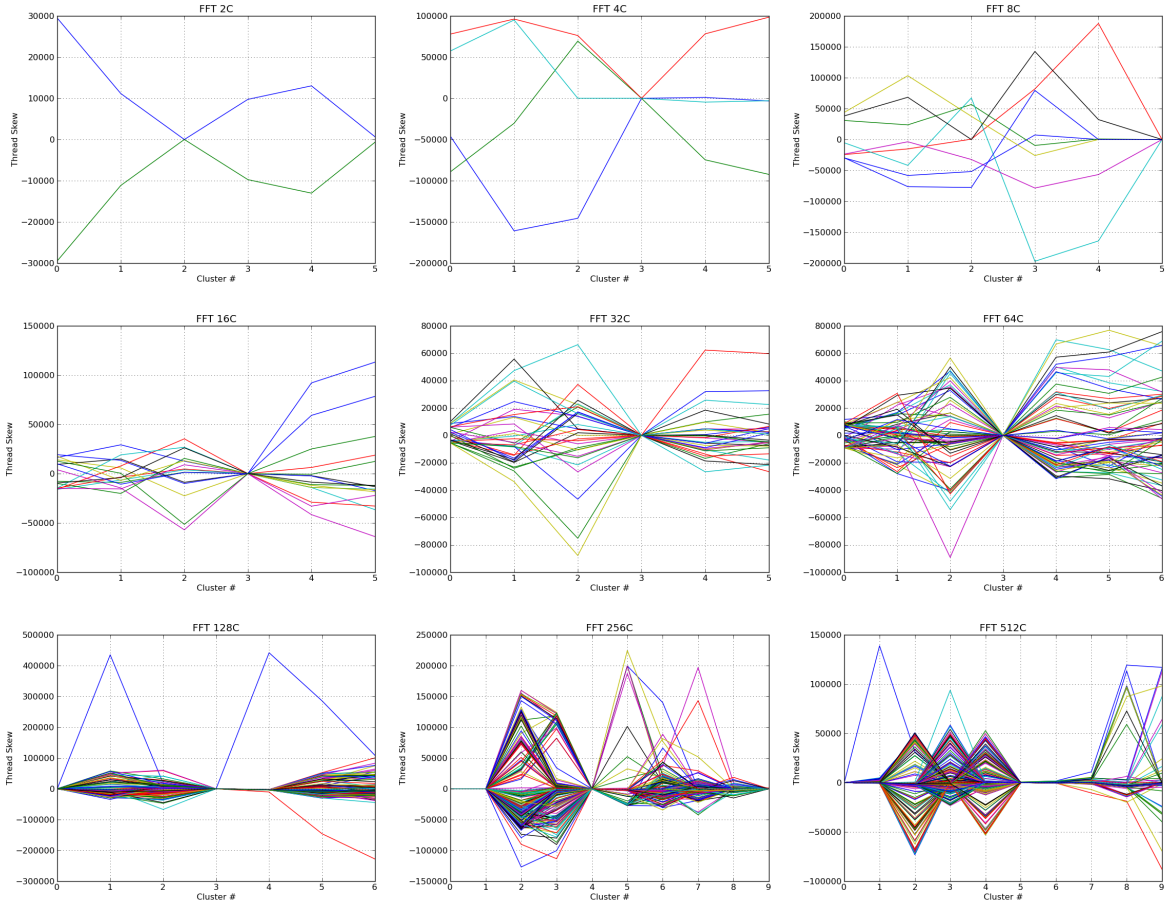


Figure 23: Thread Skew Values for *FFT* Benchmark Executions

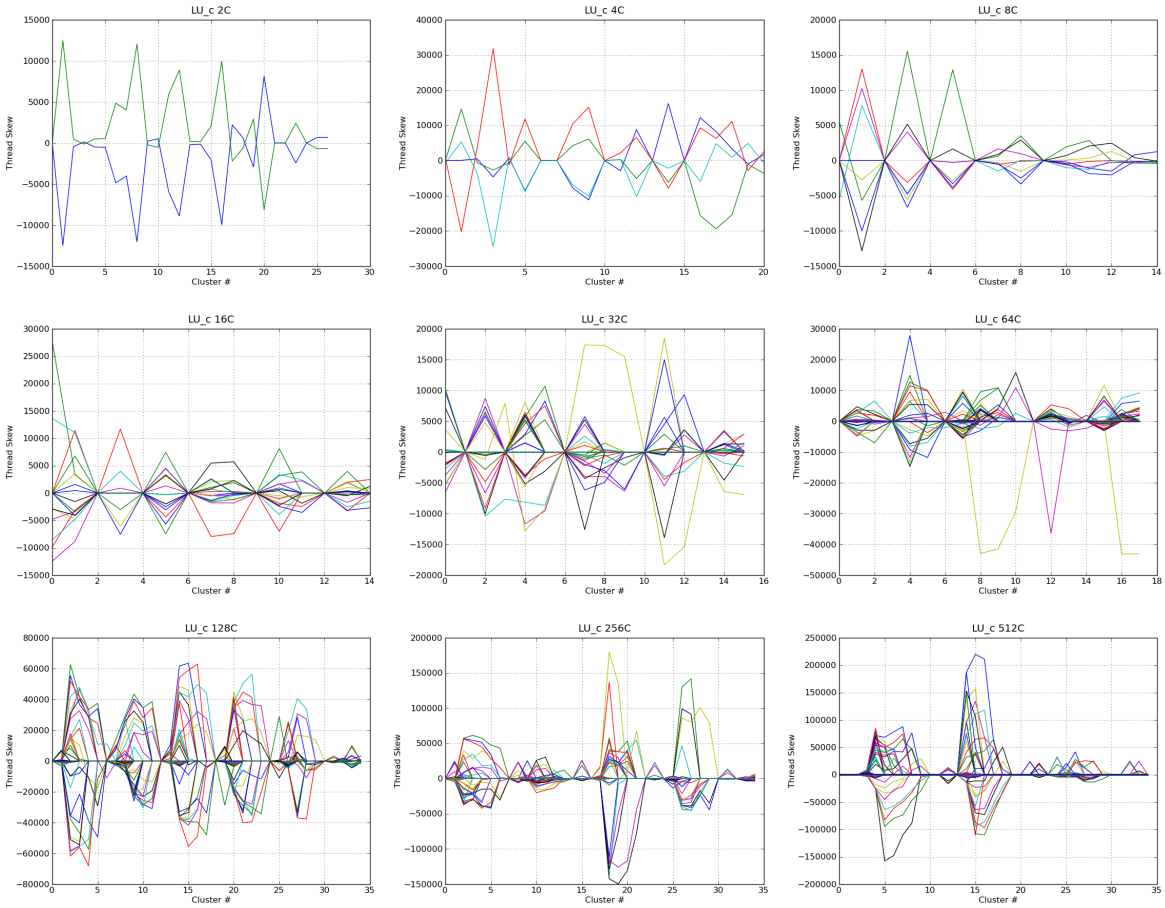


Figure 24: Thread Skew Values for *LU Contiguous* Benchmark Executions

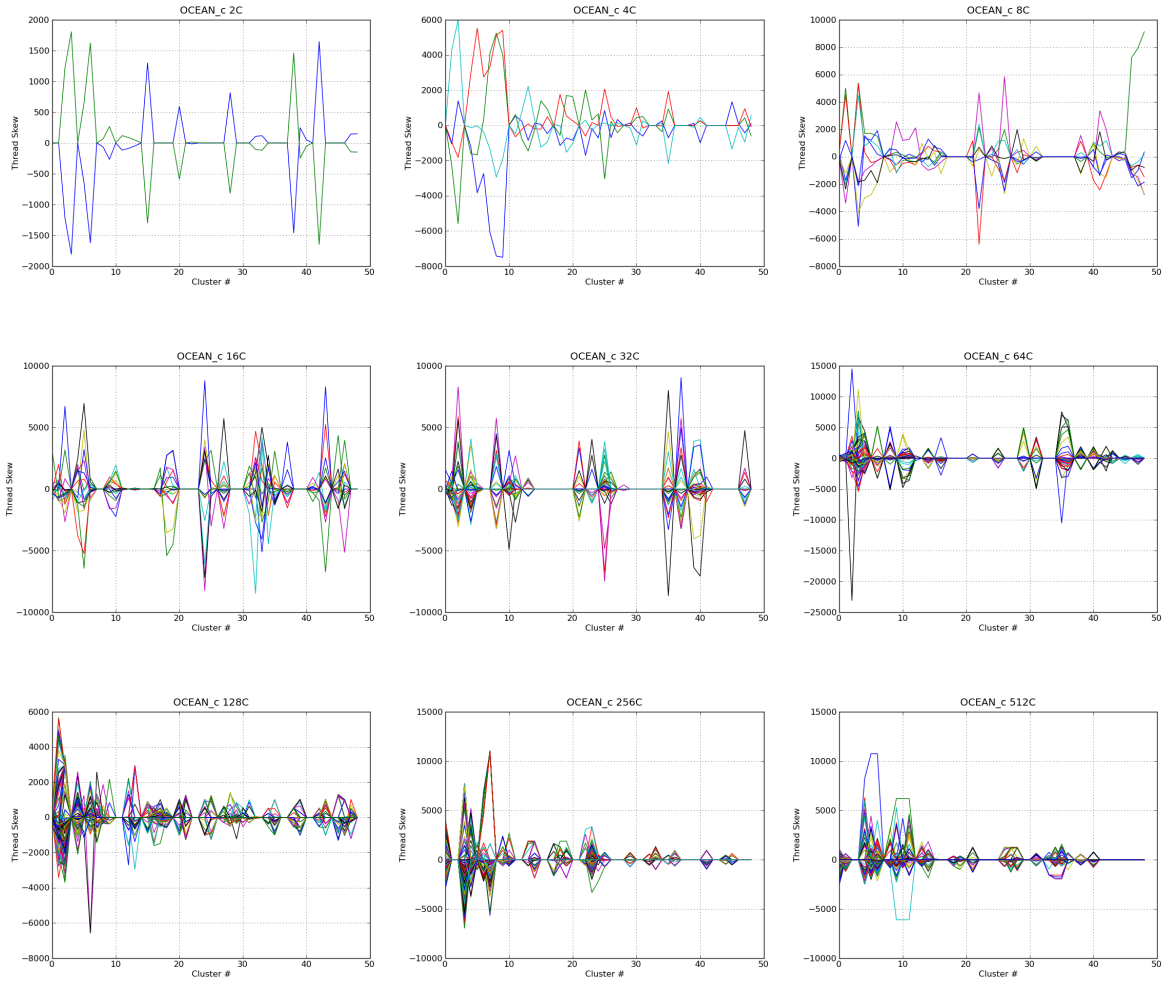


Figure 25: Thread Skew Values for *OCEAN Contiguous* Benchmark Executions

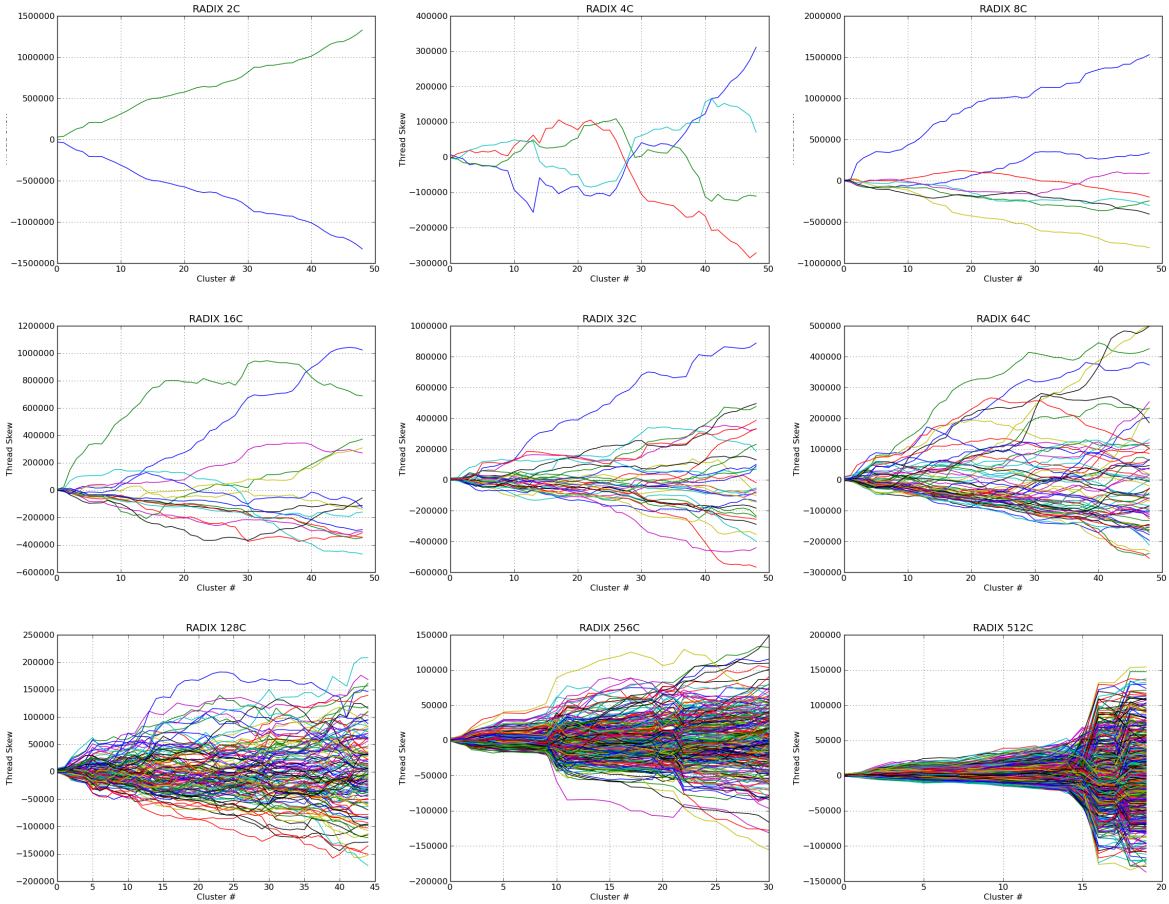


Figure 26: Thread Skew Values for *RADIX* Benchmark Executions

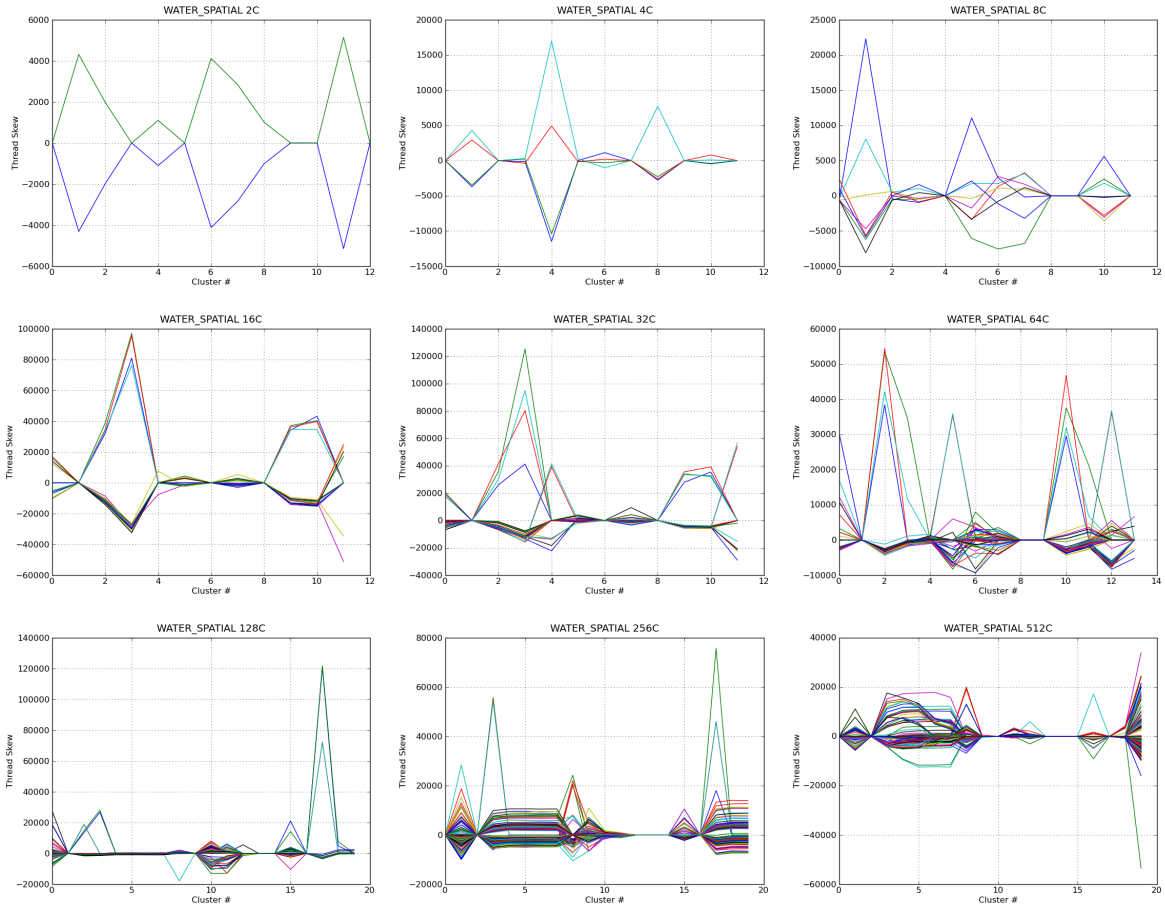


Figure 27: Thread Skew Values for *WATER SPATIAL* Benchmark Executions

5.1.5 Thread Skew Reduction

Thread skew is one of many challenges that must be overcome to reliably sample multi-threaded workloads. This section discusses a number of potential solutions which could be developed towards the reduction of thread skew.

One possible solution could involve restricting cluster placement to immediately follow barriers. This solution is similar to methods proposed by Laha et al. [49], which placed clusters at context switch boundaries. While attractive, the systematic placement of clusters can only safely be performed if the execution immediately following barriers accurately represents the entire program. Unfortunately, this is unlikely since synchronization events tend to introduce perturbations in multi-threaded executions, and are likely to contain outliers for the purposes of estimation.

Another solution could involve the artificial introduction of barriers into the program at randomly selected portions of parallel code regions. If barriers could be randomly introduced at arbitrary locations, then thread skew would be guaranteed to be zero at each of the clusters sites. However, the viability of this solution depends upon a number of criteria. First, a deadlock free compiler-assisted or other automated mechanism must be derived to automatically insert the barriers. Sample sizes in cluster sampling typically range in the order of hundreds to thousands. Without an automated method, barriers would be manually inserted through programmer directed pragmas or instrumentation of the workload under test. This would likely involve a massive effort, and would only provide estimates for a single sample. The collection of further random samples would involve manual barrier insertion- although non-parametric resampling techniques such as the jackknife or bootstrap could be used to estimate population

characteristics from a single random sample. The automated method must be performed carefully to ensure that deadlocks are not introduced through the process of barrier injection. For example, a deadlock would result if a barrier were naively introduced immediately following the acquisition of a lock. Second, the injection of barriers must not fundamentally alter application characteristics to avoid the further introduction of bias.

Other predictive modeling techniques might also prove useful in reducing thread skew, including multiple regression analysis or artificial neural networks (ANNs). Mathematical modeling techniques such as these are useful tools to establish relationships between predictor and explanatory variables. If accurate models were developed, they could be used to predict the number of instructions that must be skipped for each thread. Although models may be generated for any set of predictor and explanatory variables, difficulties arise in obtaining high coefficients of determination (R^2). The coefficient of determination measures the amount of variance observed from explanatory variables that are attributable towards variance observed from predictor variables. Models with high R^2 values (close to 1.0) indicate the majority of variance exhibited by explanatory variables has been accounted for, and are a metric for goodness of fit. One suggested method to increase R^2 for multiple regression is to iteratively add more variables to a model (while removing statistically insignificant variables). However, the use of additional variables increases the amount of training data necessary to generate the model. Obtaining training data is costly, often involving full simulation deep into the program. As more variables are added to the model, full simulation must be performed for longer periods. If the selected independent variables are interrelated

(known as multicollinearity), then it may be difficult to determine the specific effect any one variable has upon the dependent variable. Although multicollinearity does not invalidate model predictions, it may prevent or create inaccuracies in statistical data fitting. In the case of ANNs[16] the addition of variables dramatically increases the number of connections, and may result in intractable training times.

Thread skew fundamental differs from other non-sampling bias components with respect to independently and identically distributed (i.i.d.) assumptions and the propagation of errors. In single-threaded sampling, cluster measurements are approximately independently and identically distributed. Each program location has the same probability for inclusion, and cluster measurements are independent[17]. Using established state repair techniques, for example, errors encountered in one cluster are not likely to affect future measurements. If multi-threaded sampling is performed through repeated cycles of measurement and functional skipping, then the propagation of thread skew inaccuracies could become more severe as more measurements are taken (assuming no barriers have been encountered). Thus, the inaccuracies of one cluster measurement can have direct impacts upon subsequent ones, introducing a dependency between sampling units. The accumulation of thread skew errors may be avoided by launching a separate simulation for each cluster measurement. However, splitting the simulation into multiple components will dramatically increase simulation overheads, in terms of the

---

[16] This assumes a fully connected multi-layer feed-forward neural network architecture with one hidden neuron layer.

[17] Cluster measurements are actually approximately independent, given phase behavior evident in programs.

number of simulation contexts and the amount of redundant functional simulation necessary to reach clusters deep into the program.

The sufficient reduction of thread skew is a necessary condition that must be met to sample multi-threaded workloads. However, the interpretation of thread skew must be treated with care to avoid misconception. Most importantly, non-sampling bias introduced by thread skew is not proportional to sampling error. Simulations with lower thread skew do not necessarily yield estimates that are more accurate than ones with high thread skew. Measurements obtained from clusters with approximately zero thread skew tend to yield accurate estimates. As thread skew increases, sampling errors fluctuate and may increase or decrease.

5.2 Conclusion

The extension of sampling techniques to cover multi-threaded execution is a difficult problem. This chapter describes some of the challenges that must be overcome, including the circular dependence dilemma and the identification of stable performance metrics. The circular dependence dilemma captures the circular dependencies that arise when attempting to reconstruct system state at the beginning of a cluster measurement. One important non-sampling bias in multi-threaded execution is the starting positions (i.e., PCs) of all threads. This bias may be quantified through the thread skew metric, which was formally defined and measured for a number of programs. Methods to reduce thread skew were discussed, including potential solutions such as predictive modeling techniques, the restriction of cluster placement to follow barriers, and the artificial introduction of barriers. Finally, the identification of a stable performance metric is crucial towards the development of multi-threaded sampling. Without rate-based, stable

performance metrics, sampling cannot be applied, even if solutions to the circular dependence dilemma are found.

CHAPTER 6

ACCELERATING MULTI-THREADED APPLICATION SIMULATION THROUGH BARRIER-INTERVAL TIME-PARALLELISM

The design, verification, and maintenance of an architectural simulator is a complicated task [11]. Parallelization of the simulator itself raises the complexity enormously, and introduces challenges of parallel programming debugging and performance tuning. Ironically, this has led to contemporary manycore simulators executing sequentially, *even though they are used to simulate parallel systems* [7], [9], [60], [72]. This chapter presents a unique solution to parallel simulation which does not significantly increase the complexity of simulator design, verification, or maintenance.

Simulator parallelization may be divided into two classes characterized by how parallelism is extracted. The first class is *parallel discrete-event simulation* (PDES), which separates simulator tasks and state variables into a number of parallel *logical processes*. Logical processes communicate via time-stamped event messages, which are sent when other logical processes need to be notified of a particular event. PDES techniques have been leveraged to obtain high levels of concurrency in architectural simulations [62], and are promising methods to accelerate multi-threaded simulations. Several state-of-the-art simulation environments currently employ PDES [59], [62], [63]. However, PDES comes at a cost in terms of software design and computational complexity introduced by *causality errors*. Causality errors occur when a future event

erroneously affects the actions of a temporally prior event. Great care must be taken either to conservatively avoid all causality errors or to optimistically allow such errors to occur as long as they may be detected and corrected [31].

The second class of parallel simulation extracts parallelism by focusing on the decomposition of the simulation inputs rather than spatially decomposing the simulator, and is known as *time-parallel simulation*. Time-parallel simulation separates simulation inputs into a number of temporally adjacent intervals, which are then simulated in parallel [44]. In order for time-parallel methods to obtain accurate measurements, the *state-match* problem must be overcome. The state-match problem is due to cold-start effects, and is strongly related to non-sampling bias removal techniques for sampled simulation (although time-parallel simulations do not use sampling, but rather execute all intervals of the workload). Time-parallel simulations have been successfully applied to cache simulations [43], sampled processor simulation [50], as well as performance modeling [44]. Fundamentally, time-parallel and PDES approaches are orthogonal and compatible.

This work proposes a novel time-parallel based simulation methodology to rapidly accelerate the simulation of an important class of multi-threaded workloads. This work leverages the idea that barriers provide a natural, inter-thread independent point at which to split multi-threaded simulations into discrete time intervals. The proposed barrier interval simulation can also be used in conjunction with other approaches, such as PDES, to further parallelize simulation since the approaches are orthogonal and compatible. Specifically, this chapter makes the following contributions:

1. Using the thread skew metric defined in Chapter 5, this work demonstrates why barriers are useful constructs which may be leveraged to accurately parallelize single-application, multi-threaded workloads.
2. Unlike prior work which focused on process-multi-programmed or independent-task, throughput-oriented workloads, this technique is the first to apply time-parallel techniques to the simulation of single-application multi-threaded, parallel-algorithmic workloads for manycore architectures.
3. The Barrier Interval Simulation technique achieves extremely high wall-clock speedups for multi-threaded, parallel simulations with minimal losses in simulation accuracy.
4. Speedup is the most commonly used figure of merit for parallel algorithms and parallel architectures. The Barrier Interval Simulation technique provides an accurate measurement of cycle counts (a stable performance metric) that can be used to calculate speedup across multiple machine configurations.
5. This work is the first to evaluate the effectiveness of detailed warming for single-application, multi-threaded workloads, and allows the minimization of the state match problem (Section 2).

6.1 Time-Parallel Simulation

In traditional time-parallel simulation, the time axis is decomposed into a set of non-overlapping intervals. The defined intervals may not necessarily be homogenous in size, but it is beneficial if they are for load-balancing purposes (i.e., they would exhibit

higher speedups than unevenly sized intervals). Computation then consists of the following two phases: first, the *initial phase* simulates each interval with an initial guessed state (the performance measurements obtained from these initial interval simulations may therefore be inaccurate); and, the second phase, or the *fix-up computation phase*, iteratively re-simulates each of the intervals. Subsequent fix-up iterations continue until an interval's initial state matches that of the predecessor's final state (i.e., the state-matching problem [43])

This paper presents a framework based upon time-parallel simulation to speedup the simulation of single-application, multi-threaded workloads. Unlike traditional time-parallel simulation, the iterative fix-up computation phase is removed (which may limit wall-clock speedups), in place of a warm-up based approach to approximate system state. As in time-parallel simulation, the proposed technique parallelizes the input workload. This work is based on the following intuition: barriers provide a natural segmentation point to parallelize a workload.

6.2 Barrier-Interval Simulation

Barriers are an important, and commonly used synchronization construct found in many parallel algorithm implementations. They are found within the SPLASH-2, PARSEC, SpecOMP, and NAS parallel benchmark suites, among others (see Table 6). Parsec and SpecOMP barrier counts were obtained from [6] and [10]. The popularity of barrier based programs stems directly from the popular parallel programming paradigms. Directive-based languages, such as OpenMP, implicitly define barriers at parallel loop constructs. Barriers are also present in fork/join models of parallelism (e.g., CUDA). Furthermore, they are used in next-generation programming language constructs such as

Cilk's *synch* operation [20] and X10's *finish* [86] operation. Barriers are of particular importance within scientific applications, since many coarse-grained parallel programs execute in phases separated by barriers [38]. Others, such as Liu, et al. [55], leverage barriers to conserve power in CMP systems, whereas this work exploits barriers to accelerate architectural simulation.

The proposed barrier-interval simulation methodology is illustrated in Figure 28; the workload is divided into intervals defined by barrier releases, all of which are then simulated in parallel. The input workload is instrumented to identify, at runtime, barrier release events to define discrete time intervals for parallelization. Barrier release events are triggered following the last thread's arrival at a barrier, when all threads are allowed to continue execution. Each workload, comprising a parallel algorithm, is functionally executed to completion to determine the number of emulated instructions before each barrier release. These functional instruction counts provide the functional fast-forwarding values necessary to begin each simulation at the appropriate barrier release event. The functional profiling of barrier interval locations is necessary only once per workload and core count, irrespective of changes to the detailed simulator. Every interval is then simulated in parallel with a specified warm-up length. If a warm-up of W instructions were desired before an interval occurring at instruction I, fast-forwarding would be performed for I-W instructions. Detailed warming simulation continues until the first barrier release, where simulator statistics are reset. Execution of the interval then commences until the subsequent barrier release, which terminates the interval.

Table 6: Commonly-used Benchmark Suites that contain Barriers

SpecOMP		SPLASH-2		Parsec	
Benchmark	*# static barriers*	*Benchmark*	*# dynamic barriers*	*Benchmark*	*# dynamic barriers*
Ammp	7	Barnes	8	Blackscholes	8
Applu	22	Cholesky	3	Bodytrack	619
Apsi	24	FFT	6	Streamcluster	129,600
Art	3	FMM	20		
Fma3d	92	LU	34		
Gafort	6	Ocean	655		
Galgel	32	Radiosity	20		
Equake	11	Radix	14		
Mgrid	12	Volrend	15		
Swim	8	Water-nsq	19		
Wupwise	10	Water-sp	19		

The extensions necessary for a sequential simulator to support barrier-interval simulation are outlined as follows. In addition to functional fast-forwarding, the simulator must be notified of barrier release events to clear system statistics and precisely terminate intervals. The clearing of system statistics is present in many simulators since many studies include a detailed warming period after the functional skipping of initialization code. Warm-up can be applied either before or after an interval's starting point. However, if detailed warm-up consumes instructions after an interval's starting point, then errors associated with accumulative metrics such as cycle counts grow proportionally with the amount of warm-up. Although increased warm-up prior to the starting point generally improves accuracy, it does so at the expense of speedup since extra work is introduced into the simulation effort by overlapping particular instruction streams (i.e., from two or more barrier-intervals).

Barrier release events are also necessary to precisely simulate the targeted barrier-interval boundaries. Profiled interval boundaries are imprecise since they are not guaranteed to be exact locators in the instruction stream, unless fast-forwarding is

performed for all previous instructions (thus reproducing the profiled thread schedule). Simulating instructions in full cycle-accurate detail can cause divergent thread behaviors within synchronization events, such as the number of times a thread spins in a test-and-set operation waiting to acquire a lock. Thus, potential divergent thread behaviors create unknown interval boundaries, which may only be identified at runtime.

The methodology employed by barrier-interval time-parallel simulation eliminates thread skew (see Chapter 5), since the simulated intervals are guaranteed to be at boundaries where thread progressions are known (e.g., convergence points in Figures 23 - 27). By applying detailed warm-up heuristics adopted from sampled simulation, cache state and coherence information may be reconstructed to obtain highly accurate measurements over the defined intervals, producing measurements which closely resemble those of sequential simulation. Measurements obtained from individual intervals can then be aggregated to form estimated system metrics of the simulated program. For accumulative metrics, such as simulated runtime, individual measurements can simply be summed. For rate-based metrics, system metrics can be formed through the appropriate means (e.g., harmonic, arithmetic, geometric).

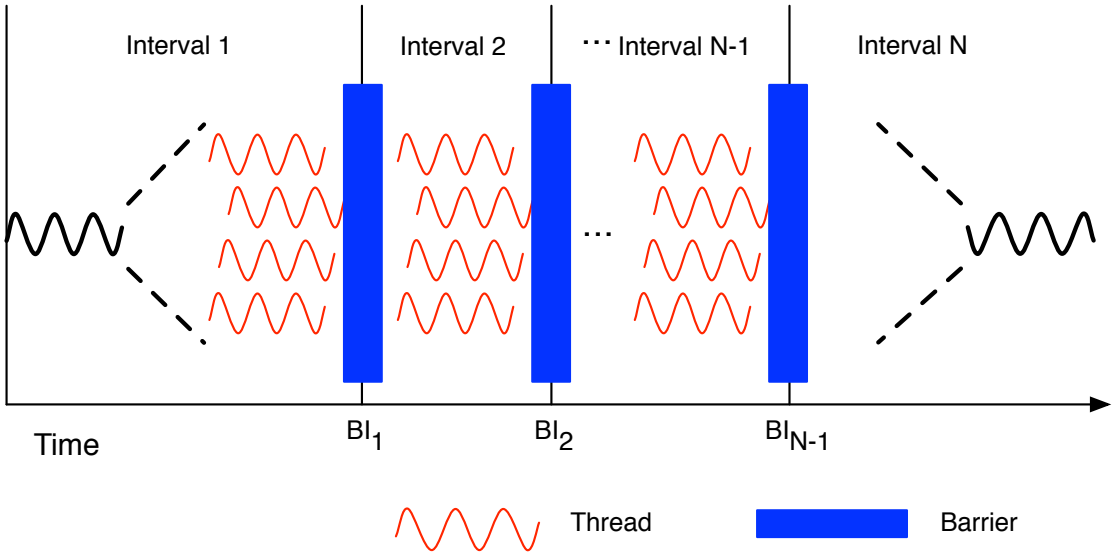


Figure 28: Barrier Interval Simulation (BIS)

The proposed parallel-time barrier-interval solution is orthogonal to the problem of stable metric identification since *all* intervals in the program are simulated in parallel, the total execution time can be reconstructed. This allows for the calculation of speedup, and speedup is the most widely used metric to assess parallel performance. It is also useful to note that higher-level or other representative metrics are also compatible with parallel-time barrier-interval simulation. Barrier-interval time-parallel simulation avoids the previously described obstacles preventing multi-threaded sampled simulation (see Chapter 5).

6.3 Experimental Methodology

Experiments in this study were conducted using the Manifold shared-memory manycore simulator, which is part of a larger, multi-agency-funded simulation framework being developed by the authors and other collaborators. The simulator is execution-driven, using the SESC front-end framework to perform functional emulation of RISC

instructions, and to provide input instructions to the detailed simulator back-end. During SESC functional emulation, threads are assigned instructions in a two-dimensional queue based upon the thread ID. During fast-forwarding, each thread is emulated by a constant number of instructions in a round-robin fashion. The detailed back-end consists of a number of architectural nodes, each containing a processor, a private L1 cache, a distributed, shared L2 cache-slice, and a network interface. The system implements a directory-based MESI coherence protocol. Nodes are connected via a network-on-chip incorporating a mesh topology that implements wormhole routing. Table 7 shows a summary of the simulation parameters. Experimental workloads consist of SPLASH-2 benchmarks cross-compiled to the target ISA using the GNU C Compiler (gcc) Version 4.2.2.

Evaluation of the barrier-interval simulation approach was performed on the following SPLASH-2 workloads: *lu contiguous*, *ocean contiguous*, *radix*, *fft*, and *water spatial*. Each workload was simulated by varying the number of cores between 1 and 512, resulting in 10 distinct simulations for each workload. For each (core count, workload) pairing, multiple detailed warming lengths were applied: none, 10k, 100k, 1M, and 10M pre-interval instructions. Although implementing fast-functional warming [84], instead of detailed warming, might produce further speedups, its use is reserved for future work. For the workloads evaluated, 181,000 simulations were performed to evaluate the trade-offs of the proposed technique in terms of speed and accuracy.

Barrier-interval simulation results are predicated upon certain assumptions. First, this work assumes that the number of threads is less than or equal to the number of cores. Second, it is also assumed threads have high affinity such that once a thread is

scheduled to a particular core, it will remain on that core. Third, the simulated workload must contain barriers in order to see performance gains. (Relaxing some or all of these constraints will be investigated in future work.)

Table 7: Architecture Parameters of the Simulated System

# Cores	1, 2, 4, 8, 16, 32, 64,128, 256, 512	Coherence / Tracking	Directory-based MESI Protocol w/ Full Presence Bits
Core Model	2-issue in-order 2 MSHRs	NOC Topology	Mesh 4-node express links
Per-node L1 Cache	32 KB set associative 4-way (WBWA) 2-cycle hit latency	NOC Router Architecture	3-stage pipeline 4 VCs / connection 2 buffers / VC
Per-node L2 Shared Last-level Cache	256KB set associative 8-way (WBWA) 8-cycle hit latency	Cache line size	64B
		Cache replacement	LRU
		Main Memory Latency	200 cycles
System L2 size	# Cores * 256KB		

6.4 Results

6.4.1 Parallel Simulation Accuracy

The accuracy of interval estimates are dependent upon overcoming cold-start effects. For multi-threaded simulation, cold-start components consist of thread skew, unknown cache, network, and directory state. Through the use of detailed warming, error components associated with unknown cache and network state were sufficiently reduced. Error results collected for individual (core count, workload) pairs for the tested warm-up lengths, and their summaries, are shown in Figure 29. Error summaries were obtained by calculating the harmonic mean of error percentages for each warm-up length. Cycle counts of the barrier-intervals were summed for the parallel simulations, and compared to

the sequential simulation using absolute relative error. On average, the error rates of the five warm-up lengths were 0.81%, 0.79%, 0.62%, 0.09%, and 0.01% for none, 10k, 100k, 1M, and 10M, respectively. Low error rates demonstrated cold-start effects associated with thread skew were sufficiently reduced, and the cache state, network state, and cache coherence information of multi-threaded workloads may be accurately approximated through the application of warm-up methods.

Larger warm-ups intuitively, and often empirically, lead to increased accuracy for interval measurements. However, certain data points, such as *lu contiguous* for 512 cores, observe higher error when a warm-up of 10k instructions is used vs. no warm-up. Error rates occasionally increased with more warm-up, but eventually converge to their expected values once sufficient warm-up is performed. One reason for this effect involves the incorrect partial warming of the caches and the on-chip network. Although system statistics are cleared at the start of an interval, network packets generated from cache misses are still in-flight when the new interval began. In general, this is desirable for reducing cold-start effects. However, in some cases high network contention caused by detailed warming can affect cache request latencies at the beginning of the interval. For example, no warm-up results in a cold network without any contention. Increasing warm-up to 10k-instructions can create a large burst of cache accesses, resulting in miss-traffic and corresponding network contention that spills into the interval execution. If warm-up is increased to 100k- instructions, however, the accesses in the shorter 10k-instruction warm-up reveal themselves to actually be cache hits due to earlier accesses in the larger 100k-instruction window. As a result, correct network contention was achieved with both the lowest and highest warm-up lengths, whereas the mid-range

warm-up length creates additional bias from incorrect miss-traffic on the network. The important observation is larger warm-ups are not always guaranteed to increase accuracy, and can even introduce additional bias.

The effect that initial state has upon measurement error is also impacted by individual thread performance. Performance is measured as the speedup relative to a single core machine. As cores are added to the simulated machine, the performance of a multi-threaded workload increases until a saturation point. Once the saturation point is reached, the addition of cores to the simulated machine begins to erode performance gains due to the increased traffic and overheads associated with thread synchronization. For the SPLASH-2 workloads, computation is divided among all the available cores. The overheads to obtain work eventually dominate useful computation, and result in system slowdown. Thus, computation performed by threads after saturation becomes increasingly non-useful. For *ocean contiguous*, the point of saturation occurred at eight cores, and had the highest error rate of all experiments. Increasing the number of cores past saturation caused long chains of requests to form, where each thread waited to access semaphores. As more threads were queued waiting to receive work, the relative importance of warm-up towards measurement accuracy diminished.

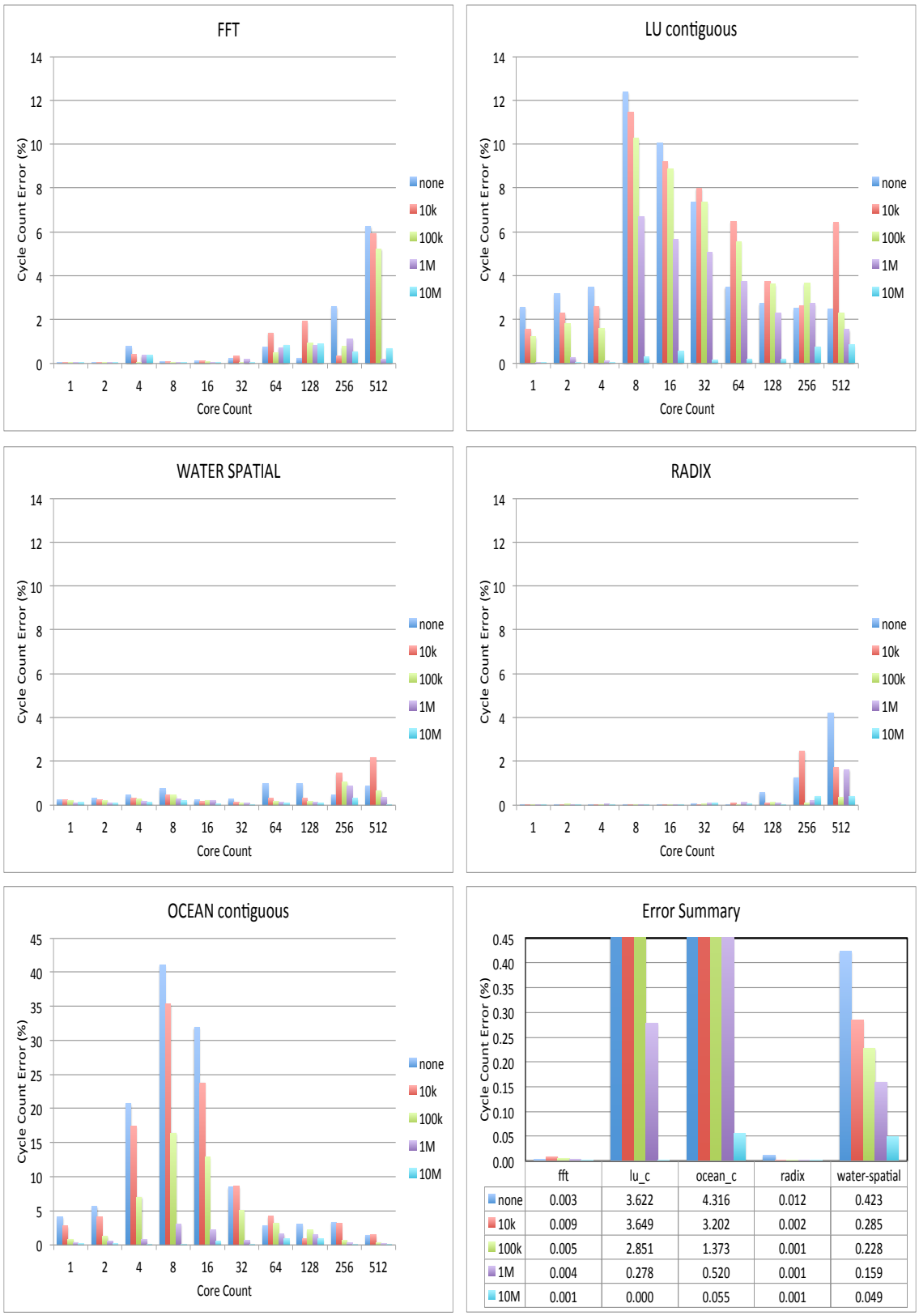


Figure 29: BIS Accuracy Measurements (per-workload and average behaviors)

6.4.2 Error Rates vs. Interval Size

In the single-threaded domain where sampling is viable, a common metric is the relationship between sampled measurements and error rates [23]. If barrier intervals are considered to be a sample of the full execution, then a similar study can be performed. Past work in the single-threaded domain found an inverse relationship between an interval's size and the measured error rates when no warm-up has been applied. The intuition behind this trend is cold-start effects are amortized across the interval. The larger the interval, the less impact that cold-start has upon measurement error. Therefore, measurements obtained from small intervals may not be reliable if warm-up is not incorporated.

The relationship between barrier interval sizes and associated error rates for the barrier-interval simulation of multi-threaded parallel workloads is also explored. To determine if the single-threaded trend between interval size and error holds for barrier intervals, the average normalized interval sizes (measured in cycles) as the number of cores increases are shown. Measurements are normalized such that the core count with the largest interval size is assigned a value of one. All experiments incorporate no warm-up. As shown in Figure 30, the interval sizes vary dramatically. In all tested workloads increasing the number of cores caused interval sizes to follow a parabolic shape, where the average size decreased to a minima before eventually increasing. The intuition behind these results is also related to the per-thread performance. Prior to saturation, additional threads cause more work to be performed in parallel, resulting in higher system performance, and a reduction in average interval sizes. After saturation, thread overheads causes additional threads to cause performance degradation of all threads, and result in

larger interval sizes. This is interesting since even without warm-up, where measurements may be the most suspect, the saturation point is correctly identified for all tested workloads. Comparisons with baseline experiments confirm that saturation occurs for all of the workloads at the smallest interval size. Saturation for *fft* occurs at 64 cores, *lu contiguous* at 16 cores, *radix* at 256 cores, *ocean contiguous* at 8 cores, and *water spatial* at 8 cores. Similar speedup limitations have been observed for SPLASH-2 in the past (see, e.g., [16]).

If multi-threaded simulations exhibit a similar relationship to interval size and error rates as single-threaded workloads, then it would have been expected that experiments containing the highest interval sizes would have the lowest interval errors, and vice versa. This was not the case, and is explained by the central limit theorem (CLT) of statistics. Average error rates for interval measurements for all workloads without warm-up are shown in Figure 31. Error rates are higher in this graph than in Figure 29 since the errors are based upon per-interval measurements rather than cumulative statistics. Even without any warm-up, increasing the number of cores causes interval errors to drop.

The distribution of interval errors with no warm-up at varying core counts for *ocean contiguous* is shown in Figure 32. Due to space constraints, only this workload is shown; however, other workloads exhibit similar behaviors. At one, two, and four cores, the distributions of errors closely follow the inverse relationship of error rates and interval sizes found in single-threaded sampling. Prior to saturation, as the number of cores increased, interval measurements begin forming clusters in the error space. These clusters of measurements decrease in size until the point of saturation, and then increase

in size as the intervals become larger. At the same time, maximum interval error rates decrease due to CLT effects. The CLT dictates the distribution of an average appears to be normal, even if the underlying distribution from which samples are taken is decidedly non-normal. The performance of individual threads may be considered as forming a distribution from which overall system performance is determined. Thus, overall system error becomes a function of component errors of the individual threads, which tends towards lower error as the number of threads increases (shown in Figure 31 and Figure 32).

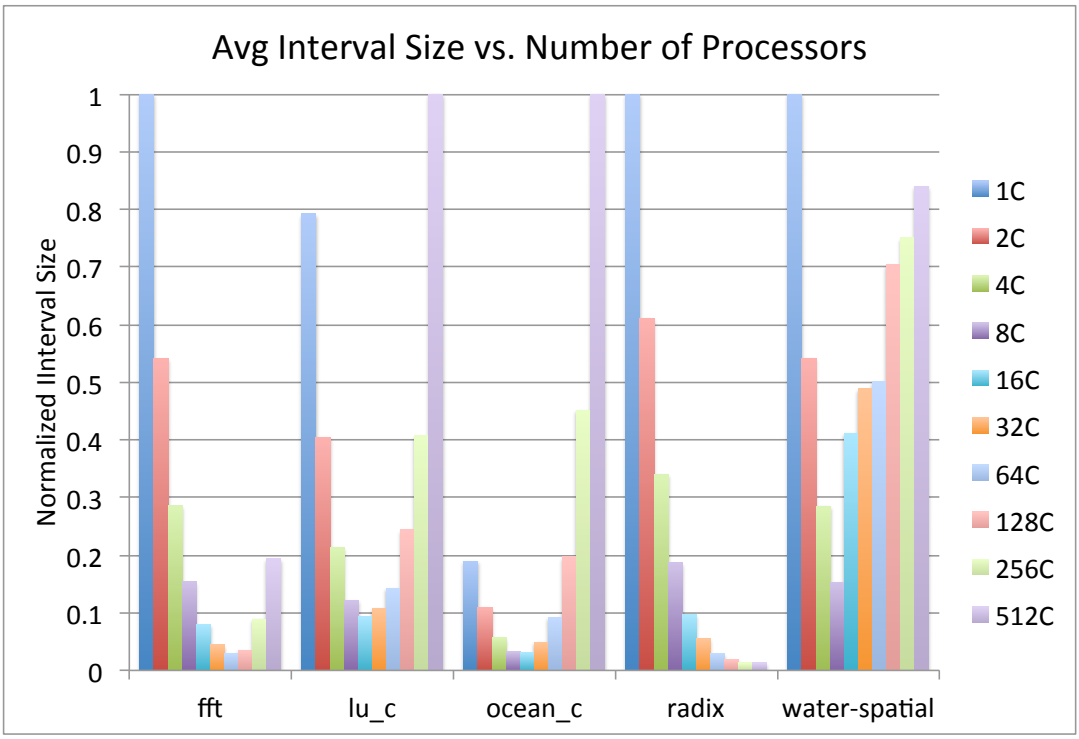


Figure 30: Average Normalized Barrier Interval Sizes

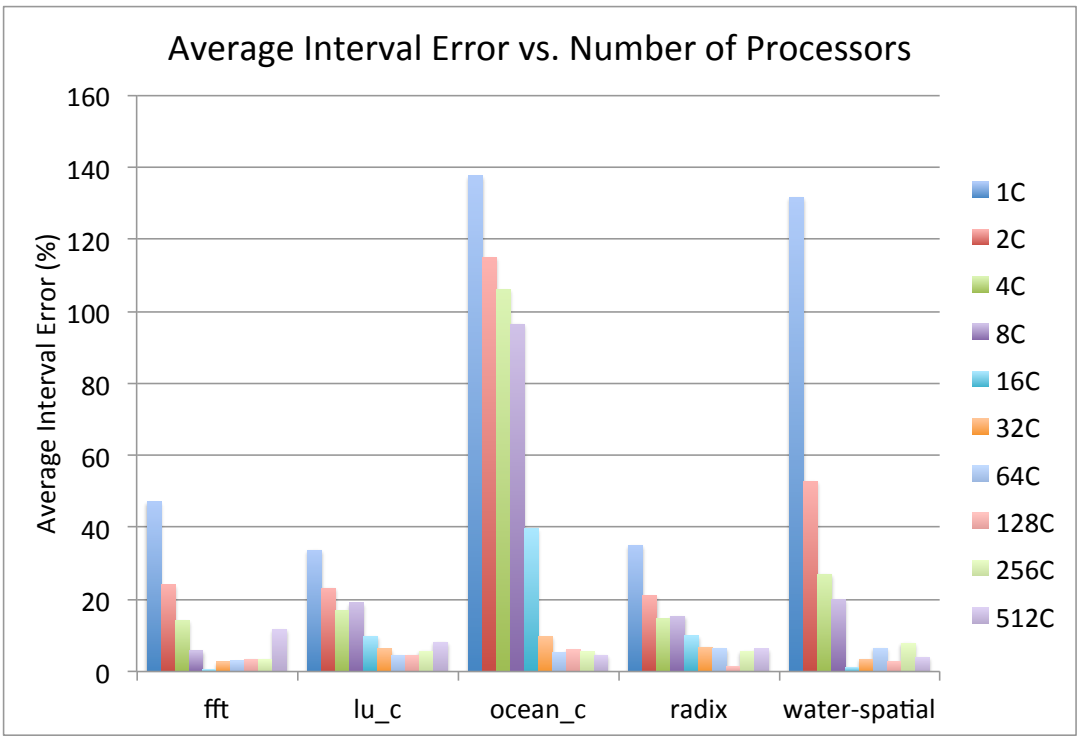
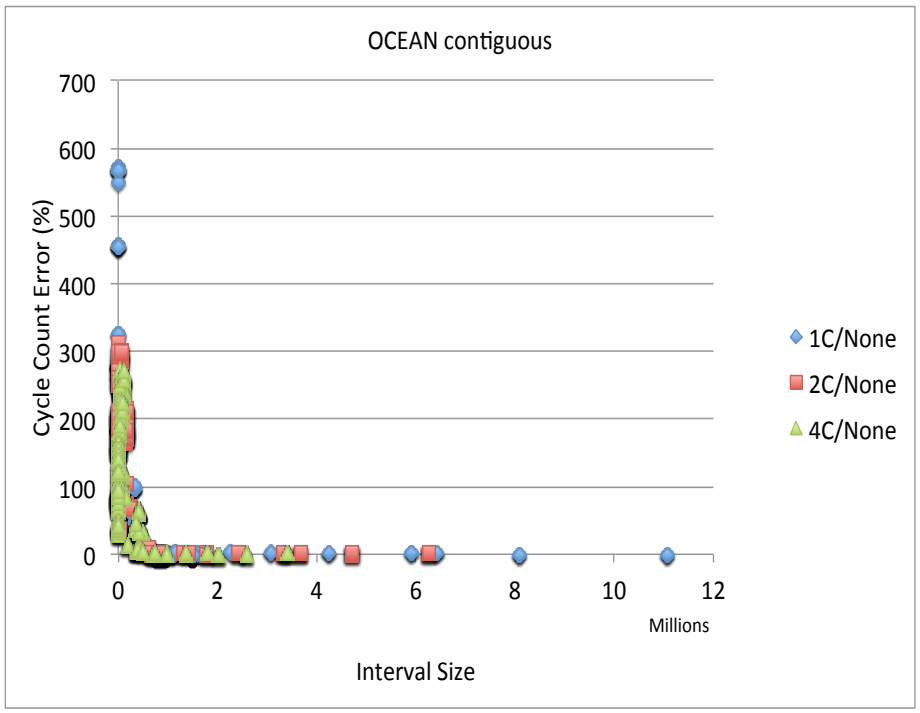


Figure 31: Average Interval Error vs. Core Counts



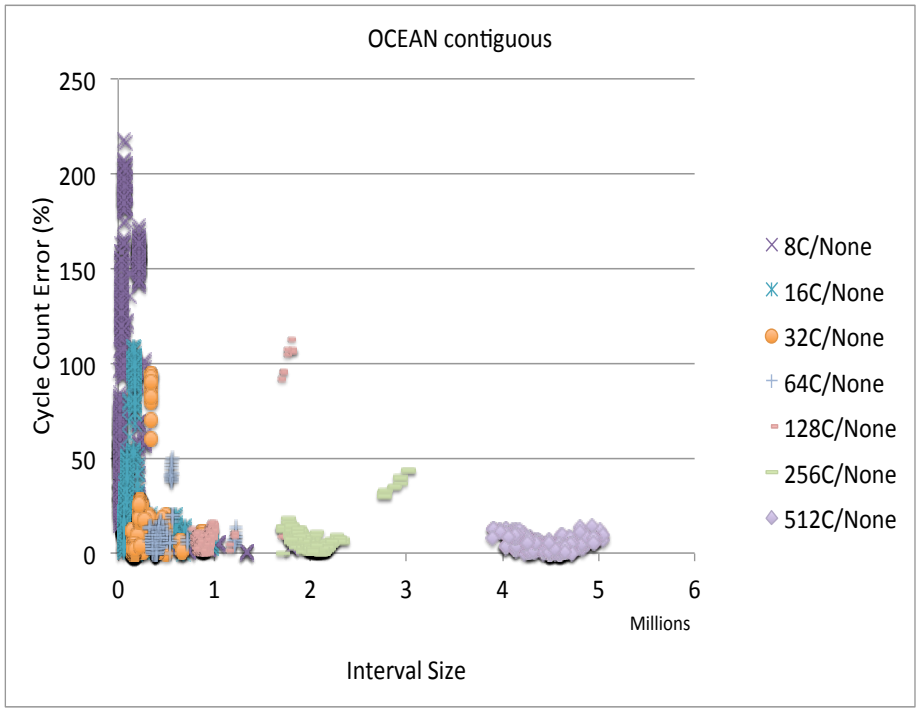


Figure 32: Interval Errors for *Ocean Contiguous* vs. Core Count

6.4.3 Parallel Simulation Speedup

For these experiments, wall-clock speedup values were calculated from repeated measurements of the sequential and time-parallel workloads. Simulations were performed on identical Intel Xeon X5450 (12MB L2Cache, 3.00 GHz, 1.33 GHz FSB) machines, with 16GB of physical memory. Since distribution outliers had large effects upon the arithmetic mean, wall-clock speedups were calculated as the ratio of median values for both the sequential and parallel simulations. Wall-clock speedup results for the five workloads are shown in Figure 33; speedup was relative to the sequential simulations. Although increasing warm-up generally improved the accuracy of interval measurements, it did so at the expense of speedup.

Measured speedups for each workload at each core count generally fluctuated as the interval size and interval homogeneity varied. Since each workload inherently contained a different number of barriers, expected speedups differed significantly from one workload to another. Table 8 shows the number of barriers contained within the simulated workloads, along with the maximum obtained speedup. The computed relative efficiency was the ratio of obtained speedup to the maximum theoretical speedup. Coefficient of Variation (CV) values were computed for each workload, which was the ratio of the standard deviation, σ, to the absolute value of the mean, $|\mu|$. As shown, there is a strong correlation between calculated CV values and relative efficiency. Lower CV values result in higher relative efficiencies. Minimum and maximum speedup values were taken from runtimes across all warm-up lengths. Even at the largest warm-up lengths, no simulation experienced slowdown over its sequential simulation.

The barrier-interval simulation methodology improved simulation times dramatically compared to their sequential simulation. On average, detailed warming using none, 10k, 100k, 1M, and 10M instruction lengths had speedups of 20.13x, 19.95x, 17.56x, 8.32x, and 3.70x, respectively. The smallest speedup of 1.22x was obtained for *lu contiguous* 10M instruction warm-up for 2 cores. The highest speedup of 596x was obtained for *ocean contiguous* with no warm-up for 512 cores.

Table 8: Relative Speedup Efficiency vs. Coefficient of Variation

Workload	Barriers	Min Speedup	Max Speedup	Rel. Efficiency	CV
fft	5	1.94x	4.10x	82%	0.3939
lu contiguous	33	1.22x	27.78x	84%	0.1025
ocean contiguous	654	1.29x	596.04x	91%	0.0564
radix	13	2.88x	4.16x	32%	0.6953
water spatial	18	2.69x	7.08x	39%	0.7070

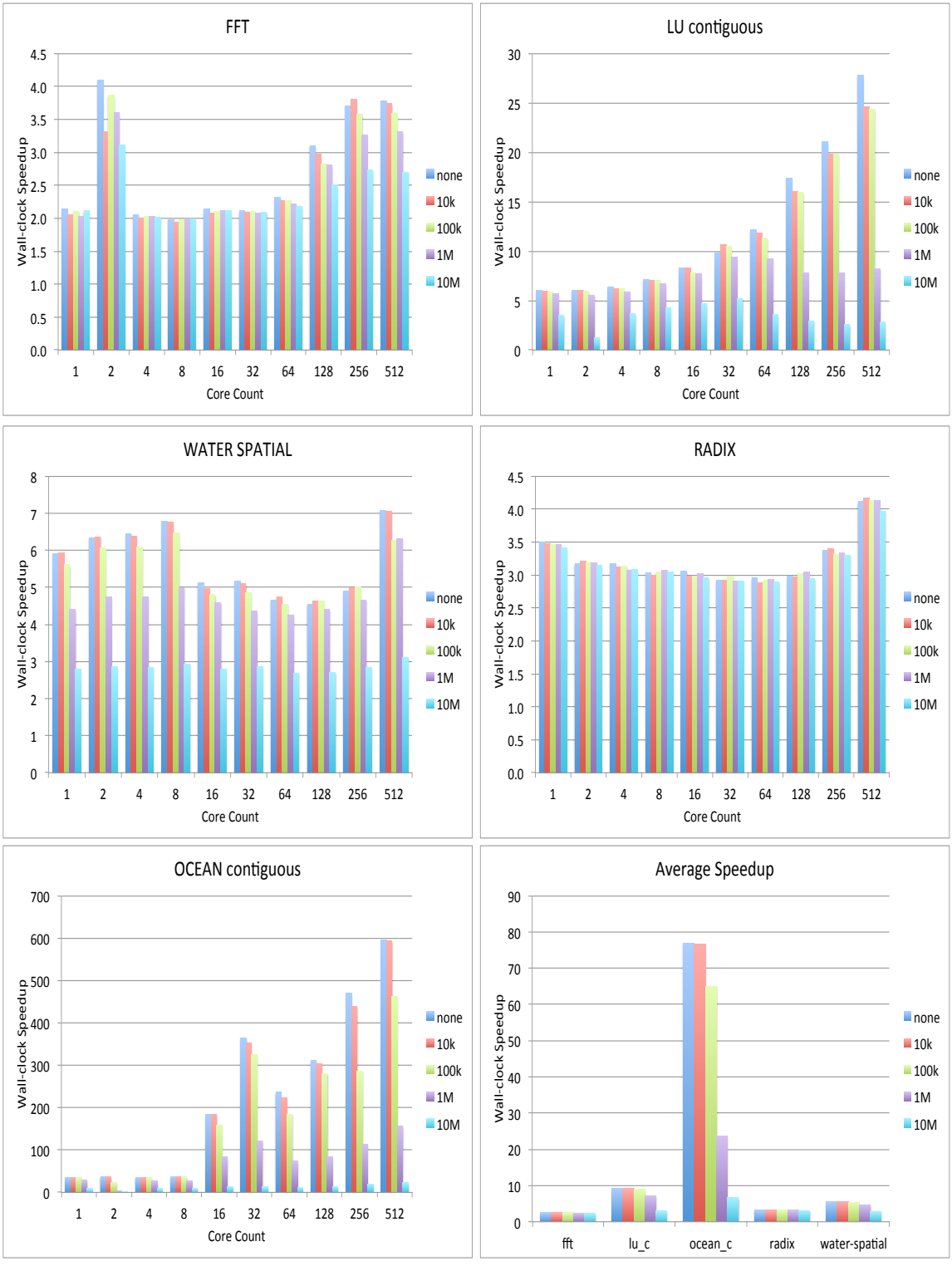


Figure 33: BIS Wall-clock Simulation Speedup Measurements

Since there were no dependencies between barrier intervals, all intervals could be simulated in parallel. Thus, the potential simulation speedup was determined by two factors: (1) the number of barriers, and therefore barrier intervals, that were contained in the workload; and, (2) the homogeneity of barrier-interval sizes. The more barriers there were in the workload, the greater the opportunity for parallelization. However, since parallelization speedup was dominated by the slowest executing interval, it was also beneficial if intervals were approximately equivalent in size. The artificial introduction of additional barriers into the workload is a possible technique that could improve the parallelization effort, however it must first be proven that additional barriers do not change fundamental properties of the simulation (both in terms of runtime characteristics and correctness), and is a topic reserved for future research. Barrier intervals could also be melded to achieve heterogeneously sized intervals, but this too is left for future research.

Barrier-interval sizes vary dramatically with the number of threads. Interval size homogeneity was measured using the coefficient of variation, which is a normalized measure of dispersion for a distribution and allows CV values to be compared across different distributions. Distributions with CV values greater than 1.0 are considered high-variance, and those below 1.0 are considered low-variance. For each experiment, the CV was calculated using the interval sizes measured in cycles. As expected, lower CV values corresponded to higher speedups. For example, for *lu contiguous*: 512 cores had a CV of 0.10 with a speedup of 27.8x; and, 2 cores had a CV of 1.45 with a speedup of 6x. CV values exhibited an inverse relationship with observed speedup for all tested workloads. Interestingly, CV values for all workloads were the smallest at the highest core counts

where interval homogeneity was improved, despite increased interval size caused by thread saturation.

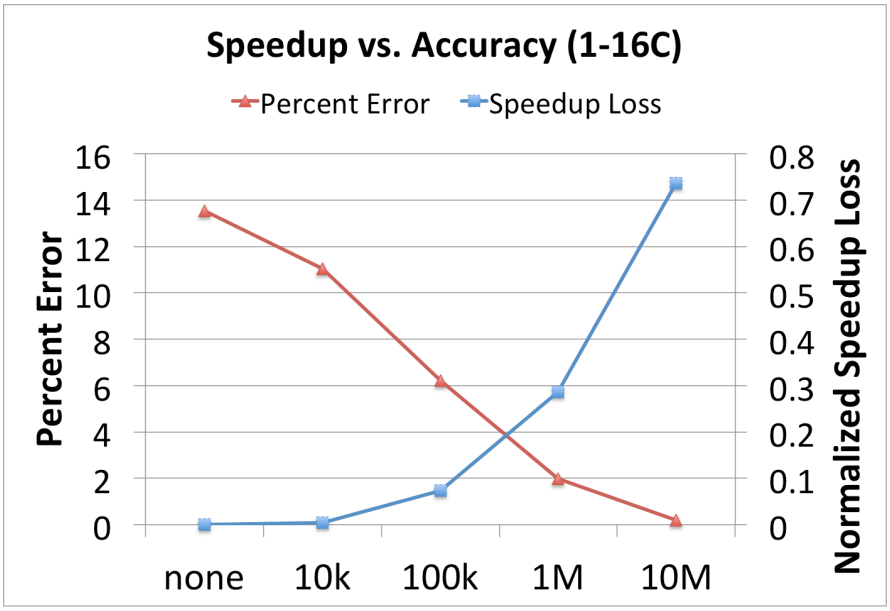


Figure 34: Accuracy and Speedup Losses vs. Warm-up (1 to 16 cores)

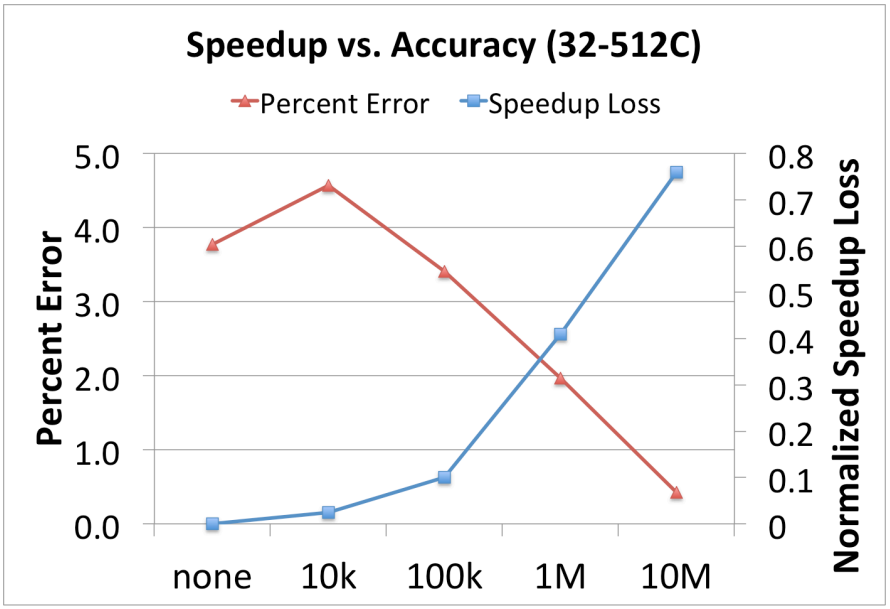


Figure 35: Accuracy and Speedup Losses vs. Warm-up (32 to 512 cores)

Larger warm-up generally resulted in increased accuracy, but rapidly diminished speedup opportunities for certain workloads. Workloads with fewer barriers (i.e., *fft*, *radix*, and *water-spatial*) were more robust towards speedup losses, and could incorporate larger warm-ups without significant penalties in performance. Since speedup losses were more prevalent in workloads containing high numbers of barriers, an analysis of *lu contiguous* and *ocean contiguous* was performed to show the speedups lost due to increased warm-up, and are shown in Figure 34 and Figure 35. The normalized speedup loss refers to the percentage of speedup (relative to no warm-up) eroded by increased warm-up. Since error rates significantly differed for these workloads at the various core counts, error and speedup values were classified into two groups: 1 to 16 cores (Figure 34) and 32 to 512 processors (Figure 35). Although higher core counts generally exhibit lower error rates even in the absence of warm-up, certain outliers exhibited non-negligible error rates (see, FFT at 512 cores). Thus, a conservative estimation of the necessary warm-up to obtain extremely high levels of accuracy resulted in a recommendation of 1M pre-interval instructions. At this warm-up length, the maximum error rate for all tested workloads was 6.7%, with an average error rate of 0.09%. Although a warm-up of 1M instructions diminished attainable speedup between 28% and 41%, the actual performance loses were not as severe if a limited context environment is assumed.

6.5 Speedup with Limited Contexts

Previous speedup calculations assumed an infinite number of physical machines are available to fully parallelize all barrier-interval simulations. For workloads with few barriers, this may be a reasonable assumption, but for workloads with many barriers (e.g.,

water spatial) this assumption is impractical. To determine potential speedups given restricted computational resources, speedups were calculated using a greedy scheduling algorithm. Barrier intervals were scheduled in descending order based on the interval's size. Figure 36 and Figure 37 show attainable speedups given limited contexts when the scheduling algorithm was applied to individual benchmarks. Results in Figure 36 were based on eight cores using 1M warm-up. Comparison with Figure 33 demonstrates limited-context speedups converged towards the speedup calculated with infinite contexts. Furthermore, the number of available contexts necessary to converge upon the theoretical value was surprisingly small for most workloads. At 16 contexts, all workloads except *ocean contiguous* converged to the maximum speedup. Figure 37 shows attainable speedups of the various warm-up lengths when all workloads where scheduled simultaneously. Results in Figure 37 were based on 64 core simulations with all tested warm-up lengths. With only 16 contexts and a warm-up of 1M instructions, the tested workloads were simulated 3x faster. This required only nine additional contexts, since sequential simulation of the workloads would require five machine contexts. When all workloads were considered, simulation speedup was limited by *radix,* which contained the largest barrier interval. The exclusion of *radix* from Figure 37 resulted in a maximum speedup of 15x with 256 contexts.

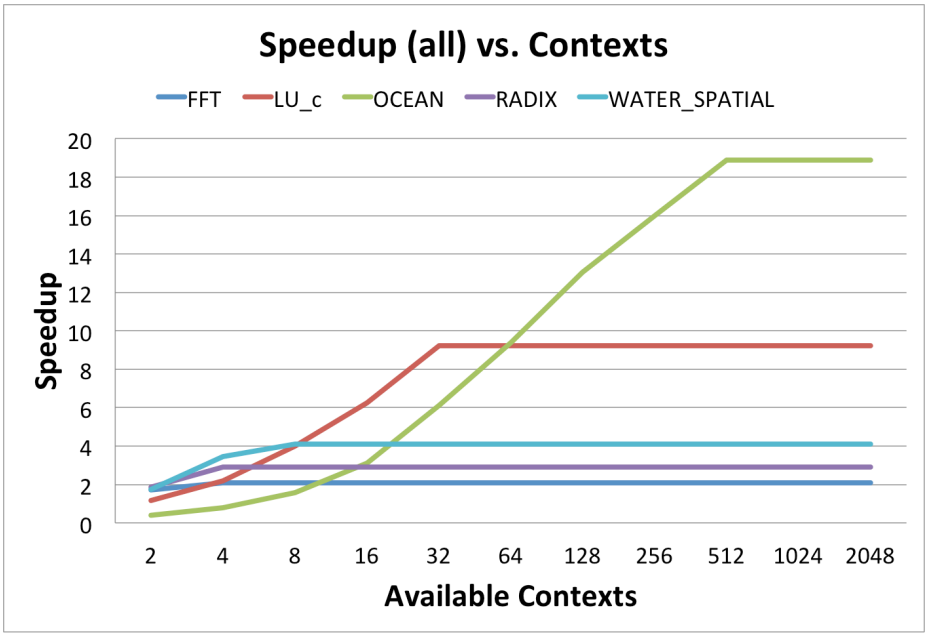


Figure 36: Limited Context Environment for each Workload

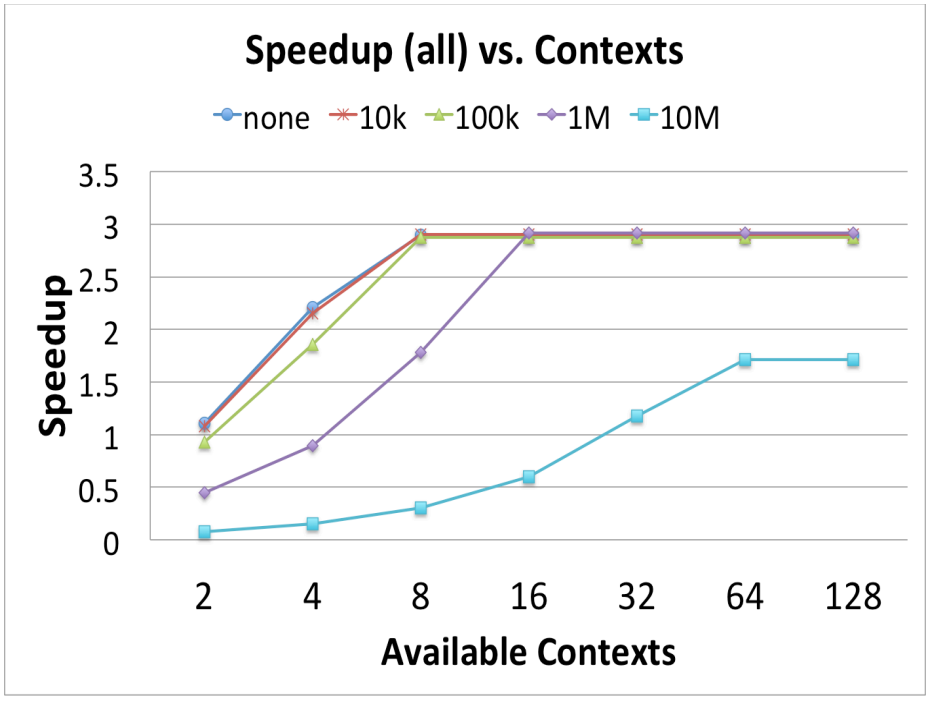


Figure 37: Limited Context Environment for the entire Suite

6.6 Related Work

Most strategies for accelerating simulation are only applicable to single-threaded applications. Identification of representative simulation points [70], benchmark subsetting [40], statistically sampled approaches [12], [13], [50], [77], [84], reduced workload input sets [46], and statistically synthesized benchmarks [1], have all been used with great success. However, the advancement of processor designs has outpaced simulation techniques. As the number of cores increase, various challenges create circular dependencies making it difficult to reconstruct simulator state, as explained below.

System performance is a combination of individual thread executions, which depend upon and are affected by system state. Thread interactions occur implicitly through shared resources (e.g., cache thrashing) or explicitly through synchronization. Race conditions due to resource locking may not be predictably modeled unless detailed state information regarding cache contents, coherence state, and network contention are known. For example, consider the common practice of skipping initialization code at the beginning of a benchmark to some later point in time. For single threaded systems, the uncertainty after functionally skipping instructions can be accurately reconstructed [13], [29], [84], [85]. For multi-threaded workloads, the approach still works provided fast forwarding occurs over a purely serial code region or fast-forwarding terminates immediately following a barrier release. However, arbitrarily fast forwarding over regions consisting of parallel-thread executions may result in system state being unknown, and the progression and precise interleaving of threads is unknown as well. The approximation of system state, therefore, is dependent upon individual thread

progressions, and thread progressions are dependent upon system state, resulting in the circular dependence previously discussed. Functional executions of these workloads result in incorrect early execution of threads, and may lead to performance measurements that differ significantly from the full execution.

6.7 Conclusion

In this chapter, a novel simulation acceleration strategy was presented to rapidly simulate certain important classes of multi-threaded, parallel-algorithm, applications with minimal losses in accuracy. The strategy can be readily implemented by architects to obtain good speedups, at low cost. Using time-parallel barrier-interval simulation, wall-clock runtimes of a number of SPLASH-2 simulations were sped up by 13.94x on average, with a maximum speedup of 596x. These speedups were obtained using a technique which can be incorporated into a number of simulation environments, including PDES based approaches. By exploiting barriers, challenges associated with the circular dependence dilemma that currently hinder the applicability of other uniprocessor accelerative techniques were avoided. Additionally, the relationship between error rates associated with state-loss obtained from interval measurements in a multi-threaded context was investigated, which could be applied towards other time-parallel or even sampled simulation domains. These results showed for parallel workloads with barriers, dramatic simulator performance gains are possible, thus shortening the design process and enabling larger workloads and input sets to be simulated efficiently.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this dissertation, various methods to accelerate microarchitectural simulation based upon statistical sampling principles were presented. The various challenges that currently face the architectural community which currently prevent the accurate and reliable sampling of single-application, multi-threaded workloads were also presented.

7.1 Contributions

The reverse state reconstruction (RSR) warm-up method was presented as an accurate and fast non-sampling bias removal technique [13]. While functionally skipping between clusters, the data necessary for reconstruction were recorded. After functional skipping had completed and the next cluster reached, processor state was approximated by functionally applying data in reverse order. By processing state in reverse order, non-sampling bias could be effectively reduced without functionally applying every record, as required by full functional warming. Although full functional warming was extremely accurate, many instructions were ineffectual and could be omitted without significant effects to non-sampling bias. Since reconstruction data were recorded, the identification of ineffectual instructions was performed without any profiling information. The proposed scheme trades storage costs for simulation speed, and proposes on-demand state reconstruction for sampled simulations. RSR algorithms were developed and evaluated for cache and branch predictor warm-up. Compared to SMARTS [80], [81], [82], [84], Reverse State Reconstruction [13] achieved maximum and average speedup ratios of 2.45

and 1.64, respectively. Simulation is a tradeoff between speed and accuracy. However, RSR obtained significant speedup improvements with negligible losses in accuracy (less than 0.3%).

The single-pass sampling regimen construction algorithm [12] was presented to quickly isolate and identify appropriate sampling regimen designs. Sampling regimen design refers to the selection of an appropriate cluster size and the number of clusters for inclusion into the sample. Historically, sampling regimen design was an iterative process that required full workload simulations for error comparisons. The iterative process involved taking samples from the workload, evaluating them against the full simulation, and performing additional simulations until an appropriate sampling regimen was found. In many instances, the identification of a valid sampling regimen involved the arbitrary selection of sampling regimen parameters (e.g., trying different cluster sizes or random seeds, etc.). In contrast, the single-pass sampling regimen method allows thousands of sampling regimen candidates to be simultaneously evaluated from a single simulation. With this technique, simulation speed was increased by an average factor of 17x with a maximum increase of 73x relative to the total workload simulation. Additionally, the single-pass sampling regimen technique allows the user to effectively estimate the true workload performance and the sample error without running the entire workload.

A novel time-parallel barrier-interval simulation technique was presented to rapidly accelerate the simulation of certain classes of multi-threaded workloads. By segmenting a program into intervals delineated by barriers, simulations could be parallelized into time-discrete intervals and avoided many of the challenges currently preventing the accurate and reliable sampling of single-application, multi-threaded

workloads. Most notably, if simulation began immediately following a barrier release, then the proper thread interleavings were approximately known. The simulator modifications necessary to support barrier-interval simulation were minimal, and were likely implemented in many architectural simulators. For the workloads tested, wall-clock speedups range between 1.2x to 596x, with an average speedup of 14x. Furthermore, barrier-interval simulation allowed for the measurement of stable performance metrics such as cycle counts with minimal losses in accuracy (2%, on average). Barrier interval simulation provides architects with a fast and accurate mechanism to rapidly accelerate particular classes of manycore simulations.

7.2 Future Work

The work performed in this dissertation provides one small step of many necessary to extend sampled simulation into the single-application, multithreaded domain. Below are some of the obstacles which must be overcome to develop sampling for manycore architectures.

1. In the single-threaded domain, architects have commonly relied upon IPC (and CPI corollary) to quantify the effects of architectural modifications. In the multithreaded domain, however, IPC can no longer be used reliably. The effect is the architectural community is currently lacking a representative and stable performance metric that may be used to characterize system performance without requiring the entire program be simulated to completion. The identification of a proper performance metric is of utmost importance, since sampling cannot be extended to multi-threaded application unless an appropriate metric is discovered.

a. Possible performance metrics must be amenable to sampling, which requires that the metric have a distribution from which to sample. Metrics that consist of aggregate counters, therefore, cannot be sampled since they do not have a parent distribution. For parametric statistics, metrics are generally required to be mean-based. However, non-parametric statistics can also be used to estimate rate-based metrics.

b. Potential metrics should be generic and workload-free to facilitate widespread adoption. Some researchers have suggested that appropriate metrics should map to the specific task being performed. Application-specific metrics include measuring frames processed per second in a graphical application, or transactions per second in a database application. If the individual work is representative of the overall execution, such performance metrics may be estimated through sampling. However, application-specific metrics suffer certain critical disadvantages. First, simulators would be required to measure performance differently for each workload under test. Second, comparisons between different applications utilizing different work-specific metrics cannot easily be made. Third, high-level metrics typically contain non-trivial amounts of execution. High-level metrics could require sampling units consist of millions (or billions) of instructions and diminish the potential cost-savings of sampling. If the sampling unit size required by high-level metrics is sufficiently large, the simulation of enough sampling units (e.g., at least 30) could be as intractable as the simulation of the complete workload.

c. Some have argued that IPC may still be used if sources of non-determinism are removed from its calculation (i.e., ignoring idle times, ignoring spin-locks, ignoring system code, etc.) [2]. Unfortunately, solutions such as these are incomplete and do not solve the problem. For example, system performance is a combination of the execution of useful work and non-useful work. Although non-useful work (e.g., acquiring a lock) does not directly contribute towards the program execution it must be executed in order for useful work to be performed. The exclusion of such instructions may bias system performance measurements, and result in measurements that do not correctly track trends. Finding a representative method of salvaging IPC for multithreaded programs, in conjunction with patching differences in the Iron Law terms between runs, is one possible solution.

2. Effective warm-up methods must be developed to minimize the non-sampling bias associated with thread skew. The performance of the individual threads in multithreaded programs is dependent upon the paths taken by the individual threads. After functionally skipping during a sampled simulation, the correct relative paths of the individual threads (i.e., thread skew) are unknown. Although the determination of the exact path of all threads for any arbitrary program location is unlikely, it may be possible to derive probabilistic bounds on the effects of thread skew. Such techniques generally require multiple sampled simulations be performed for any multithreaded workload to characterize these effects.

3. Once the challenges in (1) and (2) have been overcome, a single sampled simulation strategy may be applied to general classes of multithreaded workloads (i.e., single-application multithreaded, multi-programmed, throughput-oriented and GPU applications). The advancement of a general multithreaded sampling solution will also enable researchers to investigate and develop a host of warm-up techniques for state repair.

7.3 Conclusions

Architectural simulation is a necessary and important tool in the design of computer systems; it is also the limiting reagent. Since the complete simulation of realistically sized workloads is intractable, researchers commonly shorten runtimes using a variety of techniques. Such techniques must be based upon sound statistical principles in order for the results of simulation to be reliable. Without employing statistical rigor, simulation trends and outcomes may not represent reality. Sampled simulation is one technique to reduce simulation times that strictly adheres to fundamental statistical sampling principles. Unfortunately, the adoption of sampled simulation by the greater architectural community has not yet occurred. In almost every other discipline, researchers commonly incorporate and rely upon statistical theory. As researchers and designers increasingly rely upon small overall portions of execution to determine system characteristics, it is believed that computer architecture simulation will increasingly adopt statistical principles in system evaluation. This dissertation has contributed towards that goal.

REFERENCES

[1] Ajay, J., Lieven, E., Bell, Jr., R. H., and John, L. K. *Distilling the Essence of Proprietary Workloads into Miniature Benchmarks*. ACM Transactions on Architecture and Code Optimization (TACO), 2008.

[2] Alameldeen, A. R., and Wood, D. A. *IPC Considered Harmful for Multiprocessor Workloads*. Micro, IEEE, 2006.

[3] Alameldeen, A. R. and Wood, D. A. *Addressing Workload Variability in Architectural Simulations*. International Symposium on High-Performance Computer Architecture (HPCA), 2003.

[4] Arvind, Asanovic, K., Chiou, D., Hoe, J. C., Kozyrakis, C., Lu, S.-L., Oskin, M., Patterson, D., Rabaey, J., Wawrzynek, J. *RAMP: Research Accelerator for Multiple Processors – A Community Vision for a Shared Experimental Parallel HW/SW Platform*. Technical Report UCBICSD-05-1412, September2005.

[5] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., et al. *The Landscape of Parallel Computing Research: A View from Berkeley.* Technical Report No. UCB/EECS-2006-183, 2006.

[6] Aslot, V. *Performance Characterization of the SPEC OMP Benchmarks.* Masters Thesis. Purdue University, 2002.

[7] Austin, T., Larson, E., and Ernst, D. *SimpleScalar: An Infrastructure for Computer System Modeling*. IEEE Computer, vol. 35, no. 2, pp. 59–67, 2002.

[8] Barr, K. C., Pan, H., Zhang, M., and Asanovic, K. *Accelerating Multiprocessor Simulation with a Memory Timestamp Record*. International Symposium on Performance Analysis of Systems and Software (ISPASS), 2005.

[9] Bellard, F. *QEMU, A Fast and Portable Dynamic Translator*. USENIX Annual Technical Conference, 2005.

[10] Bienia, C., Kumar, S., Singh, J. P., and Li, K. *The PARSEC Benchmark Suite: Characterization and Architectural Implications*. International Conference on Parallel Architectures and Compilation Techniques (PACT), 2008.

[11] Black, B. and Shen, J. P. *Calibration of Microprocessor Performance Models*. IEEE *Computer*, *31*(5), 59-65. 1998.

[12] Bryan, P. D., Conte, T. M. *Combining Cluster Sampling with Single Pass Methods for Efficient Sampling Regimen Design.* International Conference on Computer Design (ICCD), 2007.

[13] Bryan, P. D., Rosier, M. C., and Conte, T. M. *Reverse State Reconstruction for Sampled Microarchitectural Simulation*. International Symposium on Performance Analysis of Systems and Software (ISPASS), 2007.

[14] Burger, D. C., and Austin, T. M. *The Simplescalar Toolset, version 2.0*. Computer Architecture News, 25(3):13-25, 1997.

[15] Cain, H. W., Lepak, K. M., Schwartz, B. A., and Lipasti, M. H. *Precise and Accurate Processor Simulation*. Workshop on Computer Architecture Evaluation using Commercial Workloads, 2002.

[16] Chauhan, A., Sheraw, B., and Ding, C. *Scalability and Data Placement on SGI Origin*. Technical Report TR98-305, 1998.

[17] Chiou, D., Sunwoo, D., Kim, J., Patil, N., Reinhart, W. H., Johnson, D. E., and Xu, Z. *The FAST Methodology for High-Speed SoC/Computer Simulation*. International Conference on Computer-Aided Design (ICCAD), 2007.

[18] Chung, E. S., Nurvitadhi, E., Hoe, J. C., Falsafi, B., and Mai, K. *PROToFLEX: FPGA-Accelerated Hybrid Functional Simulator*. International Parallel and Distributed Processing Symposium (IPDPS), 2007.

[19] Chung, E. S., Nurvitadhi, E., Hoe, J. C., Falsafi, B., and Mai, K. *A Complexity-Effective Architecture for Accelerating Full-System Multiprocessor Simulations using FPGAs*. International ACM/SIGDA Symposium on Field Programmable Gate Arrays, 2008.

[20] Cilk-5.2 Reference Manual. http://supertech.lcs.mit.edu/cilk. Oct 2012.

[21] Conte, T. M., Hirsch, M. A., and Hwu, W. W. *Combining Trace Sampling With Single Pass Methods for Efficient Cache Simulation*. IEEE Transactions on Computers, 1998.

[22] Conte, T. M. *Systematic Computer Architecture Prototyping*. PhD Thesis, University of Illinois, Urbana, Illinois, 1992.

[23] Conte, T. M., Hirsch, M. A., and Menezes, K. N. *Reducing State Loss for Effective Trace Sampling of Superscalar Processors*. International Conference on Computer Design (ICCD), 1996.

[24] Covington, R. C., Madala, S., Mehta, V., Jump, J. R., and Sinclair, J. B. *The Rice Parallel Processing Testbed*. ACM SIGMETRICS Conference on Measurement and Modeling of Computer System, 1988.

[25] Crowley, P., and Baer J.-L. *On the Use of Trace Sampling for Architectural Studies of Desktop Applications.* SIGMETRICS, 1999.

[26] Desikan, R., Burger, D., and Keckler, S. W. *Measuring Experimental Error in Microprocessor Simulation*. International Symposium on Computer Architecture (ISCA), 2001.

[27] Eeckhout, L., Bosschere, K. D., and Neefs, H. *Performance Analysis through Synthetic Trace Generation*. International Symposium on Performance Analysis of Systems and Software (ISPASS), 2000.

[28] Eeckhout, L., Eyerman, S., Callens, B., and Bosschere, K. D. *Accurately Warmed-up Trace Samples for the Evaluation of Cache Memories.* In High Performance Computing Symposium (HPC), 2003.

[29] EeckHout, L., Luo, Y., Bosschere, K. D., and John, L. K. *BLRL: Accurate and Efficient Warmup for Sampled Processor Simulation*. The Computer Journal, 2005.

[30] Fu, J. W. C., and Patel, J. H. *Trace Driven Simulation using Sampled Traces*. Hawaii International Conference on System Sciences (HICSS), 1994.

[31] Fujimoto, R. M. *Parallel Discrete Event Simulation*. Simulation Conference Proceedings, 1989.

[32] Genbrugge, D., and Eeckhout, L. *Statistical Simulation of Chip Multiprocessors Running Multi-program Workloads*. International Conference on Computer Design, 2007.

[33] Haskins, J. W., and Skadron, K. *Minimal Subset Evaluation: Rapid Warm-up for Simulated Hardware State.* International Conference on Computer Design (ICCD), 2001.

[34] Haskins, J. W., and Skadron, K. *Memory Reference Reuse Latency: Accelerated Sampled Microarchitecture Simulation*. International Symposium on Performance

Analysis of Systems and Software (ISPASS), 2003.

[35] Haskins, Jr., J. W. and Skadron, K. *Accelerated Warmup for Sampled Microarchitecture Simulation*. ACM Transactions on Architecture and Code Optimation, 2005.

[36] Henry, G. T. *Practical Sampling*. Newbury Park, CA: Sage Publications, 1990.

[37] Iyengar, V. S., Trevillyan, L. H., and Bose, P. *Representative Traces for Processor Models with Infinite Cache.* International Symposium on High-Performance Computer Architecture (HPCA), 1996.

[38] Jeremiassen, T. E., and Eggers. S. J. *Static Analysis of Barrier Synchronization in Explicitly Parallel Programs*. International Conference on Performance Analysis and Compilation Techniques (PACT), 1994.

[39] Joseph, P. J., Vaswani, K., and Thazhuthaveetil, M. J. *Construction and Use of Linear Regression Models for Processor Performance Analysis*. International Symposium on High-Performance Computer Architecture (HPCA), 2006.

[40] Joshi, A., Phansalkar, A., Eeckhout, L., and John, L. K. *Measuring Benchmark Similarity Using Inherent Program Characteristics*. IEEE Transactions on Computers, 2006.

[41] Karkhanis, T. S., and Smith, J. E. *A First-Order Superscalar Processor Model*. International Symposium on Computer Architecture (ISCA), 2004.

[42] Kessler, R. E., Hill, M. D., and Wood, D. A. *A Comparison of Trace-sampling Techniques for Multi-megabyte Caches*. IEEE Transactions on Computing, 1994.

[43] Kiesling, T. *Approximate Time-Parallel Cache Simulation*. Simulation Conference Proceedings, 2004.

[44] Kiesling, T., and Pohl, S. *Time-Parallel Simulation with Approximative State Matching*. Workshop on Parallel and Distributed Simulation (PADS), 2004.

[45] Kihm, J. L. and Connors, D. A. *Statistical Simulation of Multithreaded Architectures*. International Symposium on Modeling Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2005.

[46] KleinOsowski, A. J., Flynn, J., Meares, N., and Lilja, D. J. *Adapting the SPEC 2000 Benchmark Suite for Simulation-Based Computer Architecture Research*. Workload Characterization of Emerging Computer Applications, 2001.

[47] Kodakara, S. V., Kim, J., Lilja, D., Hsu, W.-C., Yew, P.-C. *Analysis of Statistical Sampling in Microarchitecture Simulation: Metric, Methodology and Program Characterization.* IEEE International Symposium on Workload Characterization (IISWC), 2007.

[48] Krishnan, V., and Torrellas, J. *A Direct-Execution Framework for Fast and Accurate Simulation of Superscalar Processors*. International Conference on Parallel Architectures and Compilation Techniques (PACT), 1998.

[49] Laha, S., Patel, J. A., and Iyer, R. K. *Accurate Low-cost Methods for Performance Evaluation of Cache Memory Systems*. IEEE Transactions on Computing, 1988.

[50] Lauterbach, G. *Accelerating Architectural Simulation by Parallel Execution*. Hawaii International Conference on System Sciences (HICSS), 1994.

[51] Lawton, K., et al. Bochs. http://bochs.sourceforge.net, Oct 2012.

[52] Lee, B. C., and Brooks, D. M. *Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction*. SIGOPS Operating Systems, 2006.

[53] Lee, B. C., Collins, J., Wang, H., and Brooks, D. *CPR: Composable Performance Regression for Scalable Multiprocessor Models*. International Symposium on Microarchitecture (MICRO), 2008.

[54] Lepak, K. M., Cain, H. W., and Lipasti, M. H. *Redeeming IPC as a Performance Metric for Multithreaded Programs*. International Conference on Parallel Architectures and Compilation Techniques (PACT), 2003.

[55] Liu, C., Sivasubramaniam, A., Kandemir, M., and Irwin, M. J. *Exploiting Barriers to Optimize Power Consumption of CMPs*. International Symposium on Parallel and Distributed Processing, 2005.

[56] Liu, W. and Huang, M. C. *EXPERT: Expedited Simulation Exploiting Program Behavior Repetition.* International Conference on Supercomputing (ICS), 2004.

[57] Liu, L., and Peir, J. C*ache Sampling by Sets*. IEEE Transactions on VLSI Systems, 1993.

[58] Luo, Y., and John, L. K. *Efficiently Evaluating Speedup Using Sampled Processor Simulation.* IEEE Computer Architecture Letters, 2004.

[59] Lv, H., Cheng, Y., Bai, L., Chen, M., Fan, D., and Sun, N. *P-GAS: Parallelizing a Cycle-Accurate Event-Driven Many-Core Processor Simulator Using Parallel Discrete Event Simulation*. Workshop on Principle of Advanced and Distributed Simulation (PADS 2010).

[60] Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., and Werner., B. *Simics: A Full System Simulation Platform*. IEEE Computer, 2002.

[61] Mangione-Smith, W. H., Abraham, S. G., and Davidson, E. S. *Architectural vs.*

Delivered Performance of the IBM RS/6000 and the Astronautics ZS-1. Hawaii International Conference on System Sciences (HICSS), 1991.

[62] Miller, J. E., Kasture, H., Kurian, G., Gruenwald, C., Beckmann, N., Celio, C., et al. *Graphite: A Distributed Parallel Simulator for Multicores*. International Symposium on High Performance Computer Architecture (HPCA), 2010.

[63] Mukherjee, S. S., Reinhardt, S. K., Falsafi, B., Litzkow, M., Huss-Lederman, S., Hill, M. D., Larus, J. R., et al. *Wisconsin Wind Tunnel II: A Fast and Portable Parallel Architecture Simulator*. Workshop on Performance Analysis and its Impact on Design (PAID), 1997.

[64] Noonburg, D. B., and Shen, J. P. *A Framework for Statistical Modeling of Superscalar Processor Performance*. International Symposium on High-Performance Computer Architecture (HPCA), 1997.

[65] Nussbaum, S. and Smith, J. E. *Modeling Superscalar Processors via Statistical Simulation*. International Conference on Parallel Architectures and Compilation Techniques (PACT), 2001.

[66] Oskin, M., Chong, F. T., and Farrens, M. *HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Designs*. International Symposium on Computer Architecture (ISCA), 2000.

[67] Owen, D. L. and Goldsmith P. L. *Statistical Methods in Research and Production*. Essex, England, Longman Group Limited, 1986.

[68] Patil, S., and Lilja, D. J. *Using Resampling Techniques to Compute Confidence Intervals for the Harmonic Mean of Rate-Based Performance Metrics*. Computer Architecture Letters, 2010.

[69] Penry, D. A. *The Acceleration of Structural Microarchitectural Simulation via Scheduling*. Ph.D. Thesis, Princeton University, 2006.

[70] Perelman, E., Hamerly, G., and Calder, B. *Picking Statistically Valid and Early Simulation Points*. International Conference on Parallel Architectures and Compilation Techniques (PACT), 2003.

[71] Poursepanj. *The PowerPC Performance Modeling Methodology*. ACM Communications, vol. 37, pp. 47–55, June 1994.

[72] Rosenblum, M., Herrod, S., Witchel, E., and Gupta, A. *Complete Computer System Simulation: The SimOS Approach*. IEEE Parallel and Distributed Technology: Systems and Applications, 1995.

[73] Sherwood, T., Perelman, E., Hamerly, G., and Calder, B. *Automatically Characterizing Large Scale Program Behavior*. Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2002.

[74] SPEC newsletter. http://www.spec.org, Oct 2012.

[75] Srinivasan, R., Cook, J., and Cooper S. *Fast, Accurate Microarchitecture Simulation Using Statistical Phase Detection.* International Symposium on Performance Analysis of Systems and Software (ISPASS), 2005.

[76] Van Biesbrouck, M., Sherwood, T., and Calder, B. *A Co-phase Matrix to Guide Simultaneous Multithreading Simulation*. International Symposium on the Performance Analysis of Systems and Software (ISPASS), 2004.

[77] Van Ertvelde, L., Hellebaut, F., Eeckhout, L., and Bosschere, K. D. *NSL-BLRL: Efficient Cache Warmup for Sampled Processor Simulation*. Annual Symposium on Simulation, 2006.

[78] Vengroff, D. E., and Gao, G. *Partial Sampling with Reverse State Reconstruction: A New Technique for Branch Predictor Performance Estimation.* International Symposium on High-Performance Computer Architecture (HPCA), 1998.

[79] Wenisch, T. F., et al. *SimFlex: Statistical Sampling of Computer System Simulation*. IEEE Micro, Vol 26, Issue 4, 2006.

[80] Wenisch, T. F., Wunderlich, R. E., Falsafi, B., and Hoe, J. C. *Simulation Sampling with Live-Points*. International Symposium on Performance Analysis of Systems and Software (ISPASS), 2006.

[81] Wenisch, T. F., Wunderlich, R. E., Falsafi, B., and Hoe, J. C. *Statistical Sampling of Microarchitecture Simulation.* ACM Transactions on Modeling and Computer Simulation (TOMACS), 2006.

[82] Wenisch, T. F., Wunderlich, R. E., Falsafi, B., and Hoe, J. C. *TurboSMARTS: Accelerating Microarchitecture Simulation Sampling in Minutes.* ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, 2005.

[83] Wood, D. A., Hill, M. D., and Kessler, R. E. *A Model for Estimating Trace-sample Miss Ratios*. ACM SIGMETRICS Conference on Measurement and Modeling of Computing Systems, 1991.

[84] Wunderlich, R. E., Wenish, T. F, Falsafi, B., and Hoe, J. C. *SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling*. International Symposium on Computer Architecture (ISCA), 2003.

[85] Wunderlich, R. E., Wenisch, T. M., Falsafi, B., and Hoe, J. C.. *Statistical Sampling of Microarchitecture Simulation*. ACM Transactions on Modeling and Computer Simulation, 2006.

[86] X10 release on SourceForge. http://x10.sf.net, Oct 2012.

[87] Yi, J. J., et al. *Characterizing and Comparing Prevailing Simulation Techniques.* International Symposium on High-Performance Computer Architecture (HPCA), 2005.

[88] Yi, J. J., Eeckhout, L., Lilja, D. J., Calder, B., John, L. K., and Smith, J. E. *The Future of Simulation: A Field of Dreams*. IEEE Computer, Nov. 2006.

[89] Yi, J. J., Lilja, D. J., and Hawkins, D. M. *Improving Computer Architecture Simulation Methodology by Adding Statistical Rigor*. IEEE Transactions on Computers, 2005.